Développement et évaluation d'obfuscations de protocoles basées sur la spécification

Julien Duchene

THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ FÉDÉRALE TOULOUSE MIDI-PYRÉNÉES

Délivré par :

l'Institut National des Sciences Appliquées de Toulouse (INSA de Toulouse)

Présentée et soutenue le *20/12/2018* **par :**

JULIEN DUCHÊNE

DÉVELOPPEMENT ET ÉVALUATION D'OBFUSCATIONS DE PROTOCOLES BASÉES SUR LA SPÉCIFICATION

JURY

JEAN-YVES MARION	Professeur des Universités	Rapporteur
MARIE-LAURE POTET	Professeur des Universités	Rapporteur
SARAH ZENNOU	Ingénieur de Recherche	Membre du Jury
GEORGES BOSSERT	Ingénieur de Recherche	Membre du Jury
MOHAMED KAÂNICHE	Directeur de Recherche	Membre du Jury
ÉRIC ALATA	Maître de Conférences	Directeur de thèse
VINCENT NICOMETTE	Professeur d'Université	Co-Directeur de thèse
COLAS LE GUERNIC	Ingénieur de Recherche	Co-Directeur de thèse
DAVID CAILLAT	Ingénieur	Membre invité du Jury

École doctorale et spécialité :

MITT : Domaine STIC : Réseaux, Télécoms, Systèmes et Architecture

Unité de Recherche :

Laboratoire d'analyse et d'architecture des systèmes

Directeur(s) de Thèse :

Éric ALATA, *Vincent NICOMETTE* et *Colas LE GUERNIC*

Rapporteurs :

Jean-Yves MARION et *Marie-Laure POTET*

Remerciements

Durant ces 5 dernières années, beaucoup de choses auraient pu mal se passer pour que cette thèse n'aboutisse jamais. Et malgré toutes les difficultés rencontrées, je peux enfin remercier tous ceux qui ont permis que cela se termine bien.

Je remercie ma hiérarchie qui a tout d'abord autorisé que ce projet débute puis qui l'a soutenu durant mes changements d'affectations : à DGA Maîtrise de l'Information (DGA-MI) de la Direction Générale de l'Armement ; puis au Centre d'Analyse en Lutte Informatique Défensive (CALID). Je remercie également le Laboratoire d'Analyse et d'Architecture des Systèmes (LAAS) du Centre National de la Recherche Scientifique (CNRS) qui m'a régulièrement accueilli pour effectuer mes travaux de recherche.

Je remercie grandement mes encadrants Éric Alata, Vincent Nicomette et Colas Le Guernic qui ont été présents malgré la distance pour m'aider à avancer. Ce fut un véritable travail conjoint qui a prouvé à de nombreuses reprises qu'autonomie ne veut pas dire solitude. Je remercie également Mohamed Kaâniche qui régulièrement apportait un regard extérieur neuf sur nos travaux et a sur déceler de trop nombreuses erreurs que nous pouvions laisser dans nos publications.

J'adresse également mes sincères remerciements aux membres du jury qui ont accepté de juger mon travail :

— Jean-Yves Marion, Professeur des universités à l'Université de Lorraine, Directeur du Loria ;
— Marie-Laure Potet, Professeur des universités à ENSIMAG ;
— Sarah Zennou, Ingénieur de Recherche à Aibus Group Innovation ;
— Georges Bossert, Ingénieur de Recherche à Sekoia ;
— Mohamed Kaâniche, Directeur de recherche au LAAS-CNRS ;
— Éric Alata, Maître de conférences à l'INSA de Toulouse ;
— Vincent Nicomette, Professeur des universités à l'INSA de Toulouse ;
— Colas Le Guernic, Ingénieur de Recherche à DGA-MI puis à Sekoia ;
— David Caillat, Ingénieur au CALID.

Je remercie particulièrement Jean-Yves Marion et Marie-Laure Potet qui ont accepté d'être rapportrices de ma thèse.

Je remercie également tous les membres du LAAS-CNRS avec qui j'ai passé de très bon moments, qui étaient aussi enrichissant intellectuellement que scientifiquement. Les périodes que j'ai passé parmi vous peuvent être considérés par beaucoup comme trop courtes mais elles étaient un condensé de tranches de vie et je ne suis pas certain que j'aurai pu survivre à des périodes plus longues au même rythme. Merci aussi aux doctorants et post-docs qui ont toujours su me trouver un coin de bureau et qui m'ont guidé dans des démarches simples (telles que trouver la localisation de la salle de réunion vagenyre) quand on y travaille quotidiennement mais qui sont fastidieuses les premières fois, et il y a eu de nombreuses premières fois.

Bien évidemment, j'ai passé la plupart de mon temps à traviller (pour la thèse ou pour des projets) à la DGA puis au CALID, je remercie donc tous ceux que j'ai pu cotoyer et enrichir de mon humour approximatif. Tous vous citer serait long,

fastidieux, et surtout, je suis certain d'oublier certains d'entre vous, alors je ne le ferai pas. Vous avez su me fournir des exécutoires à la recherche lors différents projets sur lesquels j'ai pu travailler avec vous, et si vous êtes

Je remercie Georges Bossert pour son aide avec Netzob et les discussions autour de la rétro-conception que nous avons pu avoir. Merci aussi à Sebastien Corfa et Rémi Félix qui ont été mes stagiaires respectivement au LAAS-CNRS et à DGA-MI. Leur travail rigoureux et sérieux a permis de gagner du temps dans ces travaux ainsi que d'explorer de nouvelles voies de recherche.

Enfin je remercie tout mes proches qui non seulement m'ont supporté (et déjà, rien que pour cela, ce sont des saints), mais m'ont aussi soutenu durant ces 5 années qui étaient un peu longues, surtout sur les 60 derniers mois. Vous avez été à mon écoute quand je pestais sur l'un ou l'autre de mes projets (si ce n'est tous). Vous avez su m'obliger à prendre du recul et du temps pour pouvoir mener à bien conjointement la thèse et le travail. Merci à vous. Et merci plus particulièrement à ma femme qui a été une présence de tous les instants.

Table des matières

Liste des Abréviations

API	Application Programming Interface, interface de programmation
ASI	Aggregate Structure Identification, un algorithme d'identification des structures de données
AST	Abstract Syntaxic Tree, arbre syntaxique abstrait
DPI	Deep Packet Inspection, inspection en profondeur des paquets
DSL	Domain Specific Language, langage de représentation de modèles spécific à un domaine
DTA	Data Tainting Analysis, algorithme d'analyse des données par propagation de teintes
GFM	Graphe de Format de Message
IDE	Integrated Development Environment, environnement de développement intégré
IDS	Intrusion Detection System, systèmes de détection d'intrusions
PDU	Protocol Data Unit, unité élémentaire de donné pour le protocole
RPC	Remote Procedure Call, appel d'une fonction distante
RTTI	Run-Time Type Information, informations de typage accessibles endant l'exécution
VM	Virtual MAchine, machine vituelle
VoIP	Voice over IP, protocole de communication audio sur la couche de transport IP
VSA	Value Set Analysis, un algorithme d'analyse des valeurs par ensemble

Introduction

Les protocoles de communication permettent à plusieurs composants d'échanger des messages de manière cohérente. Ils sont largement utilisés en informatique, dans les réseaux et en télécommunication. Un protocole peut faire l'objet d'un standard ouvert ou être tout simplement masqué aux yeux de l'utilisateur du composant. La première approche peut être choisie pour aider à la création de composants interopérables. La seconde peut être choisie par exemple quand le développeur a construit son propre protocole et ne juge pas nécessaire d'en dévoiler les détails, ou bien quand le protocole porte sur une partie sensible de la communication entre les composants, le développeur ne souhaitant pas divulguer ces détails.

La rétro-conception de protocoles consiste en l'obtention, par un analyste, d'un modèle des communications entre plusieurs composants sans disposer au préalable de tous les détails concernant le protocole. D'une manière plus générale, le but de la rétro-conception de composant est l'obtention d'informations sur le comportement ou la conception de ce composant par l'analyse du produit final. Dans la plupart des cas, cela implique d'analyser tout ou une partie du composant. Le composant peut être, par exemple, un matériel mécanique, un composant électronique, un logiciel informatique, un protocole de communication, un composant chimique ou un élément biologique. La rétro-conception de protocoles n'est pas nécessairement réalisé dans un but malveillant, elle sert divers buts qui pour la plupart sont légitimes. Par exemple, elle peut être utilisée pour l'interopérabilité, l'audit de sécurité de logiciels ou l'évaluation de conformité de l'implémentation du protocole. Cependant, elle peut aussi être employée pour des fins illégitimes tels que l'attaque d'un système ou le vol de propriété intellectuelle. Par exemple, si une société produit un jeu en ligne, ce jeu comporte des communications (régulées par un protocole) entre le client et le serveur. Un utilisateur malveillant connaissant le protocole de communication peut envoyer des informations erronées au serveur et ainsi tricher dans le jeu. Si le protocole n'est pas connu (propriétaire), la seule solution de cet utilisateur malveillant est de rétro-concevoir le protocole.

L'objectif de cette thèse est de proposer des mécanismes de protection qui visent à rendre la rétro-conception de protocole plus difficile. La protection principale d'un protocole est le chiffrement de son canal de communication. Cependant, dans le cas d'un analyste ayant accès à une application légitime qui implémente ce protocole, des outils existent pour lui permettre de contourner cette protection et effectuer la rétro-conception sur le protocole déchiffré. Les obfuscations de protocoles peuvent apporter une solution de protection complémentaire au chiffrement. Comme le souligne [Dyer 2015], les protections existantes visent surtout à contourner les différents outils de détection existants, i.e., à dissimuler le canal de communication du protocole. Ainsi, il n'y a pas à notre connaissance, de travaux visant spécifiquement à empêcher la rétro-conception de protocoles. Plus particulièrement, nous cherchons à rendre l'utilisation d'outils de rétro-conception plus difficile, obligeant l'analyste

à faire une rétro-conception manuelle.

Pour répondre à ce besoin, nous proposons un nouveau concept d'obfuscations de protocole. L'originalité de notre approche consiste à appliquer les obfuscations directement sur la spécification du protocole. En effet, si la spécification du protocole ne respecte plus les hypothèses traditionnellement considérées par ces outils de rétro-conception, la rétro-conception du protocole devient difficile. Différents types d'obfuscations atomiques sont proposées, qui peuvent aussi être composées et appliquées de façon automatique. Pour cela, le protocole est spécifié dans un *DSL* (*Domain Specific Language*) que nous avons développé, puis le code du parseur et du serialiseur est généré à partir de cette spécification. Ces codes étant générés d'après des templates comme pour les projets de serialisation, il contient les obfuscations pour le sérialiseur, et les opérations inverses pour le parseur. Ainsi, en changeant les obfuscations, il est possible de diversifier le protocole sans avoir besoin de redévelopper du code.

Dans le chapitre 1, nous présentons un état de l'art des différents outils de rétro-conception de protocole afin de déterminer les hypothèses qu'ils considèrent. Ensuite, nous comparons, dans le chapitre 2, les solutions existantes de protection de protocoles, et leurs effets sur la rétro-conception. Le chapitre 3 présente un modèle de format de message (Graphe de Format de Message, GFM) permettant l'application d'obfuscation sur le GFM. Une implémentation pratique est présentée dans le chapitre 4. Enfin, nous évaluons les performances des obfuscations dans le chapitre 5.

Ainsi, les contributions de cette thèse sont :

— Une étude comparative des outils de rétro-conception de protocole, de leurs challenges et de leur fonctionnement ;
— Un modèle de format de message permettant l'application d'obfuscations de la spécification du protocole, ainsi que des obfuscations associées ;
— Un prototype implémentant le modèle théorique validant le concept d'obfuscations automatique de spécification de protocole. Cela permet la diversification de protocoles ;
— Une évaluation pratique des obfuscations proposées.

Chapitre 1

La rétro-conception de protocoles

Sommaire

La rétro-conception de protocoles permet de retrouver la spécification d'un protocole quand celle-ci n'est pas disponible, par exemple dans le cas de protocoles propriétaires ou de protocoles hérités dont la spécification a été perdue. Depuis 2004, de nombreuses publications sont associées à des outils visant à automatiser tout ou partie de la rétro-conception de protocole. Un état de l'art de la rétro-conception de protocoles [Li 2011] a été présenté par Li *et al.* en 2011. Cependant, les informations fournies sont succinctes et ne permettent pas d'avoir une analyse synthétique et comparative des différentes approches utilisées pour l'inférence de protocole de communication. Par ailleurs, de nouvelles contributions sont apparues depuis 2011. L'état de l'art réalisé par Narayan *et al.* [Narayan 2015] en 2015 est beaucoup plus complet.

Dans [Duchêne 2017, Duchêne 2016a], nous avons effectué un nouvel état de l'art des différents outils et approches de la rétro-conception de protocoles. Il est guidé par les challenges liés à la rétro-conception de protocoles afin d'aboutir à une nouvelle classification. Tout comme Li *et al.* et Narayan *et al.*, nous nous intéressons aux contributions qui ont donné lieu à des outils. de plus, nous prenons en compte les outils de rétro-conception de structures de données qui peuvent être considérés comme des outils d'inférence du format d'un message. Ces travaux constituent la première contribution de cette thèse. Ce chapitre reprend [Duchêne 2017, Duchêne 2016a] avec les nouvelles contributions paruent depuis. Il débute par la définition des termes utiles à la rétro-conception de protocoles. Une courte présentation des domaines

d'application de la rétro-conception de protocoles précède l'identification de challenges importants qui lui sont associés. Enfin, l'évolution des outils et leur réponse aux challenges de la rétro-conception de protocoles est étudiée dans la dernière section de ce chapitre.

1.1 Définitions pour la rétro-conception de protocoles

De nombreuses études [Beddoe 2004b, Cui 2006, Newsome 2006, Caballero 2007b, Cui 2007a, Bohlin 2008] débutent par une définition des termes importants en rétro-conception de protocoles. Ces termes visent à définir à la fois les composants qui entrent en jeu dans la communication ainsi que la structure de la communication et des messages échangés. À notre connaissance, il n'existe pas de consensus concernant ces définitions. Dans ce mémoire, nous utilisons les termes les plus couramment utilisés dans la littérature.

Un *protocole* est composé de *messages*, chaque message correspondant à l'unité de données élémentaires du protocole, aussi appelée *PDU* (*Protocol Data Unit*). Les messages sont regroupés en *classes de messages* identifiés par un *type*. Un message est composé de *champs*, encodés selon un certain *format*. Les messages sont échangés selon une *grammaire*. Notons que, dans la suite, nous considérons qu'une structure de données telle qu'utilisée dans une application correspond simplement à un format de message.

Ces termes peuvent être illustrés avec le sous-ensemble du protocole TCP qui correspond à l'établissement d'une connexion, associé aux classes de messages suivantes : {`Syn, Syn-Ack, Ack`}. La grammaire du protocole spécifie que, lorsque le composant A tente d'établir une connexion avec le composant B :

1. A doit envoyer avant tout le message `Syn` ;
2. B doit répondre avec le message `Syn-Ack` ;
3. A doit finaliser l'établissement de la connexion avec le message `Ack`.

Le format du message de classe `Syn` est fourni à la FIGURE 1.1. Par exemple, le champ `flags` commence à l'offset 0x35 et possède une taille de 3 octets. Notons que cette classe de messages correspond effectivement à un ensemble de messages. La caractérisation du message en classe `Syn` est due au fait que le bit *Syn* du champ `flags` est à 1 et que les autres bits sont à 0 (i.e. le champ `flags` contient la valeur 0x2). Si ce champ doit avoir une valeur précise, tous les autres champs sont instanciés au moment de l'envoi et leur valeur peut donc changer d'une communication à une autre.

Les techniques de rétro-conception de protocoles sont habituellement classées en deux catégories : d'une part, la rétro-conception basée sur une analyse des messages échangés entre deux applications, et, d'autre part, celle basée sur une analyse d'une application utilisant ce protocole. Les messages étant généralement échangés sur le réseau, une séquence de messages échangés entre deux applications est appelée une *trace réseau*. L'analyse d'une application consiste soit en l'analyse de son code (source ou binaire) soit en l'analyse d'une *trace d'exécution* (séquence

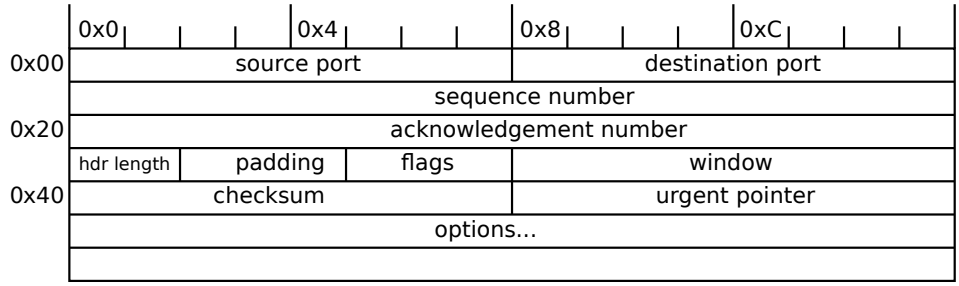

FIGURE 1.1 – Format d'un paquet `Syn` du protocole TCP, où la structure des champs est appliquée sur leur représentation en mémoire à l'adresse 0x00.

des instructions assembleur exécutées par l'application durant le traitement d'éléments du protocole à inférer). Par souci de concision, dans la suite de ce chapitre, l'inférence basée sur l'analyse d'une trace réseau sera appelée *inférence réseau* et l'inférence basée sur l'analyse d'une application utilisant ce protocole sera appelée *inférence applicative*. Il est également possible de distinguer ces techniques selon l'approche (active ou passive) utilisée pour l'inférence. L'*inférence active* stimule le système pour découvrir de nouvelles informations ou valider des hypothèses. A contrario, l'*inférence passive* est réalisée à partir d'observations capturées initialement, et donc sans nouvelle stimulation pendant la phase d'inférence. Ainsi, un outil peut pratiquer une *inférence réseau active*, une *inférence réseau passive*, une *inférence applicative active* ou une *inférence applicative passive*.

1.2 Domaines d'application de la rétro-conception de protocoles

Un des projets majeurs de rétro-conception de protocoles est le projet Samba [Samba Team 2018b] qui offre une implémentation open-source des protocoles *SMB* et *CIFS* pour des clients Linux permettant l'interopérabilité entre les systèmes Windows et Linux. En effet, la documentation d'origine n'était pas complète et en partie inexacte [Samba Team 2018a]. Cela a impliqué un lourd travail de rétro-conception. Ainsi, à ses débuts en 1992, ce projet se basait principalement sur une rétro-conception manuelle du protocole, un travail complexe, fastidieux et chronophage, dont les résultats dépendaient fortement du niveau de compétence des analystes. De plus, pour rester pertinent le projet a dû suivre les nombreuses évolutions du protocole. Toutefois, l'**interopérabilité** n'est qu'un des domaines concernés par la rétro-conception de protocoles.

La **simulation de protocoles réseau** [Leita 2005, Cui 2006, Newsome 2006, Caballero 2007a] est également un domaine pour lequel la rétro-conception peut être utile. Elle permet dans ce cas de créer des simulateurs de services réseau. Les intérêts de l'application de la rétro-conception à cette fin sont nombreux. Par exemple, un simulateur permet de réaliser rapidement des tests alors que communiquer directement avec le véritable composant peut être lourd voire impossible dans certains

cas. Il permet également d'analyser le comportement de tous les composants sur un réseau en menant des études statistiques. Pour finir, il permet de rejouer, dans des environnements variés, des traces réseau en adaptant le contenu des messages. Le rejeu est utile, par exemple, pour analyser une attaque réseau dans un environnement contrôlé. Plus concrètement, la rétro-conception pour obtenir un simulateur peut être utile dans le cas de pots de miels qui vont ainsi pouvoir entrer en interaction avec des attaquants et enregistrer leur comportement.

La rétro-conception de protocoles peut également être utilisée pour l'**audit de sécurité de logiciels** [Guihery 2012b, Caballero 2007b, Cui 2008]. Cette application est très proche de la précédente mais se distingue par son objectif qui est de solliciter un composant avec une grande variété de communications différentes pour vérifier que le composant se comporte normalement quelque soit le contexte de la communication. Ainsi, le modèle du protocole peut être utilisé pour fabriquer des fuzzers intelligents qui permettent de tester la robustesse de l'implémentation du protocole. Ces fuzzers génèrent des messages à destination du composant en modifiant certains champs des messages. Les messages qui déclenchent une vulnérabilité peuvent être traduits en signatures et ajoutés à un système de détection d'intrusion, par exemple.

L'**analyse de protocoles de malware** [Bossert 2011, Caballero 2009, Caballero Bayerri 2010] s'appuie sur la rétro-conception. En effet, bon nombre de malware, comme les bots, utilisent des protocoles pour communiquer avec leur serveur de *Command & Control*. La rétro-conception de ces protocoles peut être particulièrement utile dans la mesure où elle peut permettre d'identifier des informations cruciales concernant la localisation du maître du botnet, une date, un type d'attaque imminente, ses cibles, etc. Elle permet d'anticiper sur l'occurrence proche d'attaques et ainsi d'améliorer la réponse à incident.

Enfin, la rétro-conception peut être utile pour le **test de conformité de protocoles réseau**. Il s'agit de vérifier qu'un logiciel implémente correctement un protocole réseau dont les spécifications sont connues. La rétro-conception permet d'obtenir un modèle du protocole, tel qu'il est utilisé par le logiciel à l'étude, et de vérifier que ce modèle est en accord avec la spécification.

1.3 Challenges de la rétro-conception de protocoles

Les défis imposés par la rétro-conception de protocoles sont nombreux et variés. Ils ne sont pas tous traités systématiquement par chaque étude et il n'existe pas de solution ultime capable de traiter à la fois tous ces défis. Cette section est destinée à énumérer ces défis, identifiés en parcourant la littérature ou identifiés par notre propre expérience dans ce domaine. Pour débuter cette section, nous présentons les étapes classiques suivies lors de la rétro-conception de protocoles. Ensuite, pour chacune de ces étapes, nous identifions un défi principal qui influence la qualité globale de l'inférence. Ces défis seront ensuite utilisés pour déterminer de nouvelles obfuscations de protocoles.

La rétro-conception ne débute pas directement par les activités de reconstruction du protocole. Ces activités sont précédées d'une identification et d'une caractérisation de l'environnement. Le bon déroulement de cette préparation dépend grandement du niveau de l'analyste, comme pour les étapes de la rétro-conception et elle est primordiale pour aboutir à une rétro-conception efficace, rapide et complète. Cependant, ces aspects ne sont pas développés dans ce manuscrit et nous nous focalisons sur les étapes de la rétro-conception elle-même. Sur la base de la connaissance de l'environnement, l'analyste peut débuter cette rétro-conception, en commençant par une étape d'observation qui consiste à mettre en place l'outillage et récupérer les données qui permettront de réaliser la rétro-conception du protocole. La seconde étape vise à pré-traiter les observations pour les associer aux différents messages du protocole. La dernière étape permet d'inférer le format des messages ou la grammaire du protocole directement à partir de ces données. Bien entendu, ces étapes ne sont pas exécutées en boucle ouverte. Autrement dit, un résultat d'une de ces étapes peut amener l'analyste à revoir sa stratégie concernant les premières étapes. De plus, l'expérience et l'intuition de l'analyste peuvent réduire l'effort dédié à certaines de ces étapes. Par exemple, la rétro-conception d'un vers qui utilise un canal *Command & Control* sur `IRC` sera réalisé rapidement par un analyste qui connaît déjà ce type de protocoles.

La Figure 1.2 illustre les étapes de la rétro-conception en mettant en évidence les sous-étapes pour lesquelles nous avons identifié des défis majeurs. La suite est consacrée à la présentation de ces étapes.

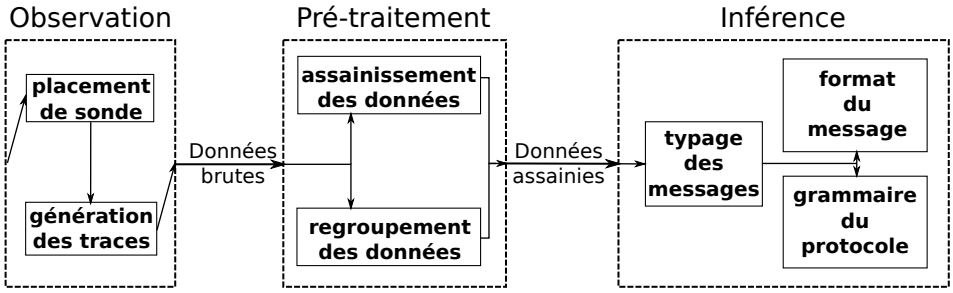

Figure 1.2 – Étapes de rétro-conception de protocoles avec les défis associés.

1.3.1 Étape d'observation

L'inférence se base sur l'obtention d'un ensemble de traces par observation du système. L'obtention de ces traces amène deux défis importants. Le premier concerne le **placement de la sonde** et le second concerne la **génération de ces traces**. Le placement de la sonde est déterminant pour l'obtention des données indispensables à la rétro-conception des protocoles. Les défis ne sont pas les mêmes pour l'approche d'inférence réseau ou pour l'approche d'inférence applicative. Pour l'inférence réseau, le placement de la sonde peut sembler évident mais il

peut aussi devenir délicat lorsque le protocole utilise plusieurs canaux de communication. Dans ce cas, il faut identifier le canal qui va véhiculer les messages liés au protocole à rétro-concevoir et il peut même être nécessaire d'envisager l'usage de plusieurs sondes, ce qui nécessite la synchronisation des différentes sondes pour conserver l'ordre des messages. Le placement doit également permettre de collecter tous les messages nécessaires à la reconstruction de tout le protocole ou de la partie intéressante. Une faiblesse au niveau de ce placement peut se traduire par des messages non observés, la reconstruction du protocole sera donc partielle. Si dans certains domaines d'application, cette faiblesse n'est pas forcément gênante, elle peut empêcher son application dans d'autres domaines où le rejeu, par exemple, est un objectif. De plus, la difficulté du placement est accentuée par l'usage de mécanismes cryptographiques qui rendent l'interprétation des traces difficile voire impossible. Il est alors nécessaire soit de casser le mécanisme cryptographique, soit de se placer aux extrémités du canal de communication, c'est-à-dire de créer une sonde placée dans l'application aux interfaces des librairies cryptographiques pour retrouver les messages en clair. Ainsi, outre le placement de la sonde, une difficulté technique s'ajoute parfois à ce défi et concerne le développement de la sonde pour l'observation.

Cette difficulté est plus poussée encore lors de l'inférence applicative qui s'appuie sur des traces d'exécution de l'application. Ainsi, la sonde doit produire la séquence des instructions exécutées par l'application lors du traitement de messages, mais aussi l'état de la mémoire du système. Plusieurs solutions existent (débogage, instrumentation, virtualisation...) mais elles doivent toutes faire face à différents défis tels que : la présence de protections contre l'analyse (contrôle d'intégrité du code, anti-debogue...), ou tout simplement le ralentissement de l'application qui peut provoquer des pertes de messages suite à un dépassement de délai.

La trace (réseau ou d'exécution) constitue la donnée principale pour rétro-concevoir le format du message ou la grammaire du protocole. La qualité de cette trace dépend du positionnement de la sonde, comme abordé dans le paragraphe précédent. Par contre, cette qualité peut également être influencée par la durée de l'observation. Évidemment, pour un même protocole, une observation courte fournira probablement une trace plus courte qu'une observation longue. Cela ne veut pas pour autant dire qu'une observation courte est insuffisante. Effectivement, il suffit simplement qu'elle ait capturé l'ensemble des états possibles de la communication et les liens entre ces états, de sorte à inclure tous les formats possibles de messages et tous les scénarios d'échange des messages possibles. Une trace trop pauvre amène à ne reconstruire qu'une partie de la spécification du format du message ou de la grammaire du protocole. La question est donc de savoir à partir de quel moment la trace est assez riche et diversifiée. Bien entendu, cela dépend de la nature des composants qui communiquent et de la fréquence des communications. Dans certains cas, l'analyste peut avoir une approche active : soit en forçant les échanges, soit en perturbant l'environnement (ce qui lui donne des indications sur la nature des échanges), soit en employant son propre composant pour communiquer.

1.3.2 Étape de pré-traitement

La phase de pré-traitement dépend largement de l'étape précédente. En particulier, aussi bien pour l'inférence réseau que pour l'inférence applicative, l'analyste n'a pas forcément pu placer la sonde à l'emplacement idéal pour l'observation et il a peut-être dû cibler un protocole sous-jacent. Du coup, les observations peuvent contenir des informations inintéressantes pour l'analyste et les messages pertinents pour la rétro-conception peuvent être :

— encapsulés dans un autre protocole ;
— découpés en plusieurs blocs transférés en plusieurs fois ;
— noyés dans une observation de multiples protocoles.

Un premier défi dans cette étape est donc de correctement nettoyer les observations pour pouvoir ensuite reconstruire les messages, cela correspond à l'**assainissement des données**. Cela implique d'être capable de discerner les informations pertinentes des informations inintéressantes. Ce défi concerne les deux types d'inférence, réseau et applicative. Toutefois, l'inférence applicative doit faire face à un défi supplémentaire qui concerne l'identification dans la trace d'exécution des structures de contrôle (sauts, boucles, tests...), qui reflètent fortement la structure encapsulée et itérative de certains messages.

Dans le cas où les messages sont transférés en plusieurs blocs, un second défi consiste donc à regrouper les traces permettant de reformer les messages pertinents pour l'analyse. Pour l'inférence réseau, par exemple, si l'observation se fait à la couche TCP, alors les paquets de segments TCP doivent être regroupés pour reconstituer des messages d'un protocole de niveau applicatif (couche OSI 7). Pour l'inférence applicative, si la trace d'exécution du traitement complet d'un message est découpée en plusieurs traces d'exécution, alors ces traces doivent être unifiées. Par ailleurs, les traces (réseau ou d'exécution) peuvent contenir des données concernant plusieurs messages, il faut donc découper les données par message. Ces deux éléments constituent le **regroupement des données**.

1.3.3 Étape d'inférence

Que ce soit pour l'inférence du format des messages ou de la grammaire du protocole, il est nécessaire de regrouper les messages en classes de messages. Ce regroupement est nécessaire pour comparer des messages ayant la même sémantique. Ce point revient à trouver une fonction permettant, à partir d'un message représenté par une séquence d'octets, de trouver le type du message, sachant que cette fonction de **typage** doit prendre en compte le degré de parenté entre les messages. Cette étape correspond à l'obtention du *vocabulaire* du protocole.

L'**inférence du format des messages** vise, à partir des messages d'un même type, à identifier leur structure et l'**inférence de la grammaire du protocole** vise, à partir de séquences de types de messages, à aboutir aux règles liées aux échanges de ces classes de messages. La réalisation de cette étape fait face à deux défis importants.

Que ce soit pour l'inférence du format des messages ou de la grammaire du protocole, il est important d'être capable d'identifier des dépendances, entre les champs d'un message ou entre les messages eux-mêmes. Sinon, le résultat de la rétro-conception risque de contenir des messages ou échanges de messages qui ne respectent pas ces dépendances.

Le résultat final de la rétro-conception est d'obtenir une spécification du format des messages et de la grammaire. Cette spécification est représentée avec un modèle. Il est donc nécessaire d'utiliser un modèle suffisamment expressif pour s'approcher au mieux de la spécification originale. Par exemple, la plupart des formats de message ou grammaires de protocoles complexes possèdent une structure arborescente ou récursive. La présence de cette structure implique qu'un message (dans le cas de l'inférence de grammaire) ou que la valeur d'un champ (dans le cas de l'inférence du format des messages) dépend des messages ou champs précédents. Ce type de dépendances est très difficile à exprimer avec des automates à états finis par exemple. Le choix du bon modèle est donc un défi majeur. Concrètement, cela revient à identifier l'outil adéquat pour l'inférence. Un mauvais choix risque d'aboutir à une spécification adaptée uniquement à représenter les observations sans capacité à généraliser. De plus, les observations sont généralement incomplètes, ainsi, les outils doivent utiliser des approches suffisamment robustes pour généraliser les modèles obtenus avec un minimum de sur-approximation.

Dans la littérature, nous avons identifié trois classes de modèles utilisés :

— **Structures énumérées** : les modèles associés sont les templates qui sont généralement utilisés pour représenter le format des messages. Ils sont par ailleurs annotés dans une étape de post-traitement pour pouvoir représenter des dépendances entre champs.

— **Structures itératives** : les modèles associés sont les langages réguliers qui sont utilisés à la fois pour le format des messages et pour la grammaire de protocole. Ils sont souvent représentés sous la forme d'automates finis déterministes ou de machines de Mealy pour la grammaire de protocole, et sous forme d'arbres pour le format des messages, où chaque nœud correspond à un champ du message. Ces arbres sont aussi annotés pour garder des dépendances entre champs. Contrairement aux templates, cette représentation permet de mettre en évidence l'encapsulation de plusieurs couches protocolaires dans le message.

— **Structures récursives** qui ne sont utilisées que pour la grammaire de protocole. Elles peuvent être représentées sous la forme d'automates à pile ou de chaînes de Markov cachées.

Si ces modèles sont issus de l'ingénierie des protocoles, ils sont adaptés (par des étapes de post-traitement) à la rétro-conception des protocoles, et au but final de cette rétro-conception (cf. section 1.2). De plus, certains modèles souffrent de restrictions liées à leur méthode d'apprentissage. Ainsi, le format BNF (Backus-Naur Form) (équivalent à des automates à pile) est utilisé par certains outils mais sa méthode d'apprentissage ne lui permet d'inférer que des structures itératives.

La synthèse de cette section est représentée dans le schéma de la FIGURE 1.2

qui reprend les trois étapes présentées ainsi que la plupart des défis associés.

1.4 Évolution des outils de rétro-conception de protocoles

De nombreuses contributions ont fait évoluer la rétro-conception de protocoles. Les outils présentés par la suite se basent principalement sur deux contributions majeures. Le premier outil, *PI Project* [Beddoe 2004b], a introduit les algorithmes de bio-informatique pour l'inférence réseau de format de messages. Le second outil, *Polyglot* [Caballero 2007b], a présenté une nouvelle technique d'inférence de format de message à partir de traces d'exécution d'une application traitant une entrée. Le tableau 1.1 présente les différentes publications associées aux outils d'inférence réseau et applicative respectivement. Les outils en gras sont ceux utilisant une approche active. Ces diagrammes montrent que ces outils se sont majoritairement concentrés sur l'inférence de format de messages, l'argument principal développé dans la littérature est qu'avant de pouvoir inférer la grammaire du protocole, il faut que le format des messages soit connu. De plus, nous constatons que l'inférence active commence tout juste à voir le jour dans ce domaine.

1.4.1 Inférence réseau

Dans [Beddoe 2004a, Beddoe 2004b], l'auteur présente l'outil, ***PI Project***[1] pour réaliser la rétro-conception du format de messages à partir de traces réseau. Cet outil laisse donc de côté l'inférence de la grammaire du protocole. Il emploie l'algorithme *Needleman & Wunsch* [Needleman 1970] pour aligner les séquences d'octets entre les messages pris deux à deux. L'alignement optimal permet d'identifier les parties communes entre deux messages qui correspondent, selon l'outil, aux mêmes champs. Ce résultat est utilisé pour construire un arbre correspondant à la classification hiérarchique des messages en fonction de leur degré de similitude (les messages les plus similaires se trouvent sur des feuilles voisines de l'arbre). L'algorithme employé est UPGMA [Nei 1983]. Par contre, l'utilisateur de cet outil doit partitionner manuellement cet arbre pour identifier les classes des messages. La majorité des outils d'inférence réseau de format de messages, parus par la suite, s'appuie sur ces travaux.

Peu après, Leita *et al.* ont proposé l'outil ***ScriptGen*** [Leita 2005, Leita 2008], qui a pour objectif de générer des scripts de simulation réseau du comportement d'un serveur, pour le pot de miel *Honeyd*. Contrairement à *PI Project*, l'inférence de la grammaire du protocole est nécessaire à cet outil pour atteindre cet objectif. Tout d'abord, *ScriptGen* opère une phase de pré-traitement correspondant d'une part au filtrage des paquets réseau pour conserver uniquement les paquets TCP des communications à simuler et d'autre part à la reconstruction des messages à partir de ces paquets. Concernant l'inférence du format des messages, l'approche

1. Code source disponible à `http://www.4tphi.net/~awalters/PI/PI.html`

TABLE 1.1 – Évolution des contributions pour l'inférence de protocole. Les outils utilisant une approche active sont mis en gras.

	Network based inference		**Application based inference**	
	Message Format	Protocol Grammar	Message Format	Protocol Grammar
2004	PI Project [Beddoe 2004b, Beddoe 2004a]			
2005	ScriptGen [Leita 2005]	ScriptGen [Leita 2005]		
2006	**RolePlayer** [Cui 2006]		FFE/x86 [Lim 2006] **Replayer** [Newsome 2006]	
2007	Discoverer [Cui 2007a]		Polyglot [Caballero 2007b] Rosetta [Caballero 2007a]	
2008			AutoFormat [Lin 2008] Tupni [Cui 2008] Prospex [Wondracek 2008] **ConfigRE** [Wang 2008]	
2009			ReFormat [Wang 2009] Dispatcher [Caballero 2009] **Fuzzgrind** [Campana 2009b]	Prospex [Comparetti 2009]
2010	ASAP [Krueger 2010]	Veritas [Wang 2010]	REWARDS [Lin 2010]	**MACE** [Cho 2010]
2011	ReverX [Antunes 2011] Netzob [Bossert 2011]	ReverX [Antunes 2011] Veritas [Wang 2011]	Howard [Slowinska 2011]	**MACE** [Cho 2011]
2012	Netzob [Guihery 2012b, Guihery 2012a]	**Netzob** [Guihery 2012b, Guihery 2012a] PRISMA [Krueger 2012]	ARTISTE [Caballero 2012]	
2013			Dispatcher [Caballero 2013]	
2014	**Netzob** [Bossert 2014b]	**Netzob** [Bossert 2014b]	**AFL** [Zalewski 2017]	
2015			ARGOS [Zeng 2015]	

employée est similaire à celle utilisée dans *PI Project*. Par contre, le partitionnement en classes de messages est réalisé automatiquement en s'appuyant sur une valeur seuil. Concernant l'inférence de la grammaire, *ScriptGen* utilise une approche basée sur l'inférence de langages réguliers. Le modèle résultant est un automate déterministe qui contient uniquement les réponses les plus fréquemment retournées par le serveur.

En parallèle à ces travaux, Cui *et al.* ont présenté leur outil ***Role-***

player [Cui 2006] qui a pour but de rejouer des communications dans un nouvel environnement, notamment pour étudier plus en détail le déroulement d'une attaque réseau. Le rejeu nécessite d'adapter une trace aux caractéristiques de l'environnement (par exemple, changement d'adresses IP, de numéro de séquencement, etc.). Il s'agit donc d'avoir une connaissance précise des champs des messages et de leurs dépendances. Pour traiter ce défi, *Roleplayer* utilise une inférence active durant laquelle des communications sont initiées par l'outil lui-même pour identifier ces dépendances. En appliquant également un algorithme d'alignement, il est capable d'apprendre des dépendances entre les champs d'un message (par exemple, la taille, les valeurs cookies, etc.) sans pour autant être capable d'identifier les encapsulations de champs. S'appuyer sur les résultats de cet apprentissage permet d'analyser plus finement les attaques qui ne contiennent pas beaucoup d'échanges.

Face aux difficultés rencontrées pour identifier correctement les champs d'un message, Cui *et al.* ont proposé une nouvelle approche dans l'outil ***Discoverer*** [Cui 2007a]. Leur inférence ne repose plus sur l'algorithme de bio-informatique *Needleman & Wunsch*. Ils supposent à présent connus les délimiteurs des champs (espaces, retours à la ligne, indentations, virgules, etc.), ce qui simplifie grandement le problème. Suite à ce découpage, l'outil enchaîne avec une classification hiérarchique et récursive des messages qui présente l'avantage de reconnaître des situations d'encapsulation des champs. Pour finir, une série d'heuristiques est appliquée pour retrouver des dépendances entre les champs.

Alors que la recherche sur l'inférence de protocole se focalisait sur l'inférence du format des messages, l'outil ***Veritas*** [Wang 2010, Wang 2011] vise principalement à construire la grammaire du protocole. Sans nécessiter le format des messages, il lui est indispensable de connaître la classe de chaque message. Pour identifier ces classes, l'outil se base sur les séquences d'octets les plus fréquemment observées dans leur en-tête (les n premiers octets). Les auteurs font ainsi l'hypothèse que l'en-tête d'un champ est localisée au début du message et que celui-ci contient des séquences d'octets caractéristiques de la classe du message. Non seulement cette hypothèse est généralement vérifiée pour faciliter l'interprétation du message, mais de plus, en procédant de cette manière, il réduit considérablement le volume des données à traiter. Les séquences d'octets fréquentes servent donc à classer les messages. Il enchaîne ensuite avec l'inférence de la grammaire du protocole. Pour cette étape, les auteurs supposent que les messages qui peuvent être émis dépendent uniquement du dernier message observé et non de tout l'historique des échanges. Le modèle construit par cet outil est donc naturellement un automate étiqueté avec les classes de messages. Notons qu'avec cette approche, *Veritas* exclut volontairement les séquences d'octets rares qui peuvent correspondre à des données ou à un comportement inhabituels du serveur. Dans ce dernier cas, la grammaire du protocole obtenue est partielle et ne représente pas tous les comportements inclus dans les échanges observés.

Pour leur part, Krueger *et al.* ont développé l'outil ***ASAP*** [Krueger 2010] pour aider l'analyse de données issues de pots de miel, la conception d'IDS (systèmes de détection d'intrusions) et l'analyse de malware. Dans ce contexte, cet outil se

focalise plus sur la classification des messages qu'à l'obtention du format exact de ces messages. Il utilise aussi un découpage des messages en champs, en fonction de délimiteurs prédéfinis pour des messages au format texte ou en séquences d'octets de taille fixe pour les messages au format binaire. Chaque message peut alors être caractérisé en fonction des champs qu'il contient et représenté sous la forme d'un vecteur. Les classes de messages sont identifiées comme étant des messages types qui constituent une base pour la génération de tous les messages observés, au sens algébrique. Au delà des choix concernant les outils statistiques et la manière de découper les messages, *ASAP* se distingue donc principalement des autres outils par l'usage d'une approche algébrique. Par contre, le format obtenu n'est pas très précis et ne met pas en évidence les liens entre les champs ni leur ordre d'apparition ou les champs qui seraient à cheval sur deux lignes ou deux séquences d'octets.

Par la suite, les développeurs de *ASAP* ont étendu leur outil en y ajoutant une capacité d'inférence de la grammaire de protocole, ce qui a donné lieu à la création de l'outil ***PRISMA*** [2] [Krueger 2012]. Cet outil emploie globalement la même stratégie que *ASAP*. Par contre, l'analyse algébrique des messages est remplacée par une classification basée sur une distance pour regrouper les messages en classes. Ensuite, l'outil analyse les séquences de classes de messages pour en déduire une chaîne de Markov cachée. L'usage de ce modèle présente l'avantage de pouvoir représenter des changements d'état du protocole qui ne se traduisent pas nécessairement par l'envoi d'un message.

À l'opposé des approches précédentes, l'outil ***ReverX*** [3] [Antunes 2011] utilise une approche unique : il traite le format des messages de la même manière que la grammaire du protocole. À l'instar de *Discoverer*, cet outil se base sur une liste connue de délimiteurs pour découper les messages en champs. Ces séquences sont assemblées sous la forme d'un arbre qui est à son tour replié sur lui-même. Ce repli aboutit à un automate contenant des boucles qui sont utiles pour généraliser le modèle à des messages non observés. Cet automate correspond ainsi aux formats des messages. La grammaire du protocole est reconstruite de la même manière, en partant des séquences de classes de messages. Cet outil offre des résultats intéressants, notamment au niveau de la généralisation du format des messages, qui constitue un défi important. Par contre, la stratégie qu'il emploie pour découper les messages en champs ne le rend pas adapté à l'analyse de messages binaires.

Pour terminer, citons l'outil ***Netzob*** [4] qui a fait l'objet de nombreux travaux différents et qui a connu de multiples évolutions ces dernières années. Il a dans un premier temps été conçu pour la modélisation de botnets [Bossert 2011]. Les auteurs utilisent une variante d'une machine de Mealy pour représenter la grammaire du protocole, en enrichissant cette machine par des informations sur le temps d'attente entre deux messages observés. Cette grammaire de protocole est apprise de manière active à partir de l'algorithme L^* [Angluin 1987]. Cet automate est ensuite rendu non-déterministe en incorporant des probabilités d'émis-

2. Code source disponible à `https://github.com/tammok/PRISMA`
3. code source disponible à `https://github.com/jasantunes/reverx`
4. Disponible en source ouverte à `https://www.netzob.org/`

sion de messages étant donné un état et un message reçu. Les techniques utilisées pour le regroupement des messages et l'inférence de leur format sont abordés dans [Guihery 2012b, Guihery 2012a]. Ainsi, l'algorithme *Needleman & Wunsch* est utilisé pour effectuer l'inférence de format de messages et la classification des messages à l'instar de l'outil *PI Project* [Beddoe 2004b]. Les auteurs introduisent de plus une heuristique nommée *orphan reduction* qui leur permet d'effectuer un alignement local entre deux messages, sans considérer l'intégralité du contenu de ces messages. Ensuite, une étape d'inférence de la sémantique des champs est réalisée, à partir d'heuristiques similaires à celles présentes dans *Discoverer* [Cui 2007a]. Ils conservent cependant dans leur format, les différents attributs inférés, tels que l'encodage, car ces informations peuvent aider un analyste à peaufiner l'inférence. Alors que l'inférence de la grammaire du protocole est basée sur une approche active, celle du format des messages est entièrement passive. Dans l'article [Bossert 2014b], ils apportent des améliorations majeures à leurs résultats en utilisant une approche active. Ils insèrent des données connues dans l'application implémentant le protocole puis recherchent ces mêmes données dans les différents messages. Ils utilisent aussi une corrélation temporelle afin de mieux associer les messages avec des actions sur l'application. Ces deux éléments leur permettent d'une part de délimiter de façon certaine des champs dynamiques, et d'autre part d'associer une sémantique aux classes de messages. Cependant, cette approche nécessite un travail supplémentaire qui peut être complexe à automatiser de la part de l'analyste. Enfin, cet article présente également une étude comparative des résultats avec ceux de *PRISMA* [Krueger 2012] et de *ScriptGen* [Leita 2005], chose qui n'avait pas été réalisée auparavant. Ces travaux ont été complétés dans la thèse de G. Bossert [Bossert 2014a], notamment en ce qui concerne la formalisation des modèles utilisés pour le format des messages et la grammaire du protocole, ainsi que le développement de techniques actives pour l'inférence de la grammaire. Cette nouvelle inférence leur permet de découper l'apprentissage de l'automate en sous automates liés à des actions sur le système. Cela lui permet de réduire drastiquement le temps d'apprentissage de la machine de Mealy. Il est à noter que l'outil *Netzob* possède un greffon d'exportation des résultats sous différents formats ainsi qu'un greffon de *fuzzing* et de simulation réseau intégré. De plus, c'est à notre connaissance, le seul outil utilisé en production et encore maintenu à ce jour. Il fait partie avec *PI Project*, *ReverX* et *ASAP/PRISMA* des rares outils dont le code source est disponible.

Synthèse

Le tableau 1.2 résume les différentes approches utilisées par les outils pour rétroconcevoir des protocoles par l'analyse de traces réseau en précisant pour chaque outil si l'approche est active ou passive, le type de mesure utilisée pour regrouper les messages, le modèle utilisé pour le format des messages et sa capacité à identifier des dépendances entre champs, ainsi que le modèle utilisé pour l'inférence de grammaire de protocole.

TABLE 1.2 – Résumé des outils d'inférence réseau. Un tiret indique que l'outil ne pratique pas ce type d'inférence. Par exemple, *Veritas* ne travaille pas sur le format des messages

Outils	**Approche**	**Classification des messages**	**Format du message**		**Grammaire du protocole**
		Mesure basée sur :	*Dépendances entre champs*	*Modèle utilisé*	*Modèle utilisé*
PI Project	Passive	Alignement + Classification manuelle	Non	Template	-
ScriptGen	Passive	Alignement + classification automatique	Non	Template	Automate fini déterministe
RolePlayer	Active	Alignement	Oui Sans encapsulation	Template	-
Discoverer	Passive	Délimiteurs + classification automatique	Non	Template	-
Veritas	Passive	Découpage récursif en champs de taille fixe + classification automatique	-	-	Automate fini déterministe
ReverX	Passive	Délimiteurs + classification automatique	Non	Automate fini déterministe	Automate fini déterministe
ASAP	Passif	Délimiteurs ou découpage + classification algébrique	Non	Template	-
PRISMA	Passive	Délimiteurs ou découpage + classification automatique	Non	Template	Chaînes de Markov cachées
Netzob	Active	Alignement + inférence active	Oui	Template	Machines de Mealy non déterministes

Cette synthèse montre que l'inférence réseau se base principalement sur :
— des techniques d'alignement de séquences pour le découpage des messages en champs et leur regroupement en classes de messages ;
— des heuristiques pour l'inférence de la sémantique des champs et des dépendances intra-messages ;
— une approche basée sur l'inférence de langages réguliers pour la grammaire du protocole.

Ce n'est que très récemment que les outils ont commencé à nettement se distinguer. Tout d'abord, l'inférence active utilisée par *RolePlayer* et *Netzob* leur permet d'améliorer le découpage en champs. Ensuite, *PRISMA* et *Netzob* prennent en compte l'état du serveur en employant des modèles probabilistes. Enfin, seul l'outil *ReverX* utilise les mêmes techniques pour traiter le format du message et la grammaire du protocole. Cela lui permet notamment de représenter le format des messages sous forme d'automates à états finis alors que les autres outils utilisent généralement de simples templates. Le tableau 1.3 résume les contributions majeures de chacun des outils ainsi que le défi de la rétro-conception de protocole auquel il répond.

1.4.2 Inférence applicative

Lim *et al.* sont les premiers à avoir présenté un outil, ***FFE/x86*** [Lim 2006], d'inférence applicative de format de messages. Le but de cet outil est d'obtenir le format des données produites par une application en employant une analyse purement statique du binaire. À notre connaissance, il s'agit du seul outil à employer cette approche. Ainsi, il n'est pas soumis au défi d'observabilité rencontré par les outils analysant une trace d'exécution. De plus, il fait partie des rares outils étudiés (avec *Rosetta*, *Replayer* et *Dispatcher*) à inférer le format des messages produits et envoyés par l'application analysée (par opposition aux messages reçus et traités). Plus précisément, il déduit, du graphe d'appel des fonctions, un automate hiérarchique représentant le format des messages. Ensuite, deux techniques d'analyse statique, *VSA* (*Value Set Analysis*) et *ASI* (*Aggregate Structure Identification*) implémentées dans l'outil *CodeSurfer/x86* [Reps 2005], sont utilisées afin d'obtenir un sur-ensemble des valeurs prises par les données en sortie. Le résultat de ces analyses est incorporé directement dans le modèle du format des messages pour le généraliser. La sortie est exprimée sous la forme d'une expression régulière qui est plus facilement exploitable par un analyste humain que l'automate hiérarchique.

En parallèle à ces travaux, Newsome *et al.* ont publié l'outil ***Re-Player*** [Newsome 2006]. Contrairement à *FFE/x86*, ils utilisent une analyse hors-ligne d'une trace d'exécution. Leur but est de pouvoir rejouer des communications, c'est-à-dire, tout comme *RolePlayer* [Cui 2006], d'identifier les champs du protocole dépendant de l'environnement. Une contribution originale de ce travail est d'avoir formalisé le problème de rejeu de communications. Pour cela, ils déduisent des messages, des contraintes identifiant les données liées à l'environnement. Pour le rejeu, ils utilisent la formule de contraintes couplée avec un solver (STP) pour générer

TABLE 1.3 – Résumé des contributions majeures de chaque outil d'inférence réseau et du challenge qui est résolu par cette contribution

Outil	**Contribution**	**Défi résolu**
PI Project	Algorithme d'inférence réseau de format de messages basé sur Needleman & Wunch avec du clustering manuel	Découpage automatique en champs
ScriptGen	Inférence de la grammaire de protocole par la théorie des langages, micro/macro clustering	Inférence de la grammaire de protocole + clustering automatique + nettoyage de traces réseau contenant du bruit
Roleplayer	Inférence active des champs dépendant de l'environnement	Rejeu de traces réseau dans un autre environnement
Discoverer	Classification hiérarchique et récursive des messages à partir d'un découpage par délimiteurs connus + heuristiques pour les relations entre champs	Encapsulation des champs + identification de dépendances telles que les tailles ou les champs de session
Veritas	Clustering basé sur l'entête des messages permettant l'inférence de la grammaire	Réduction du volume de données à traiter + inférence de la grammaire sans le format de message
ASAP	Clustering basé sur une métrique algébrique sans inférence de format de message précis	Travail sur de nouvelles métriques pour le clustering des messages
PRISMA	Utilisation de chaines de Markov cachées pour la grammaire de protocole	Représentation d'états ne se traduisant pas forcément par une émission de messages
ReverX	Inférence de format de messages par théorie des langages	Amélioration de la généralisation des messages
Netzob (2012)	Grammaire de protocole inférée de façon active + interactions utilisateur	Apprentissage de séquences de messages par approche active
Netzob (2014)	Inférence active du format des messages + corrélation temporelle pour les classes de messages	Amélioration de la précision du format des messages, réduction du temps d'apprentissage pour les machines de Mealy

les messages qui auraient été produits dans un nouvel environnement. Cependant, seules les contraintes sont déduites et le format des messages n'est pas réellement inféré. Par conséquent, même si l'outil accomplit à lui seul cette tâche, il est difficile de généraliser son usage à d'autres domaines. Son incapacité à inférer le format des messages et la grammaire du protocole implique que, contrairement à *ScriptGen*, *Replayer* rejoue exactement la même séquence de classes de messages.

Peu après, Caballero *et al.* ont publié ***Polyglot*** [Caballero 2007b] qui a une approche différente basée sur l'analyse des traces d'exécution pour l'inférence du format des messages. Cette approche et les techniques d'analyse associées sont majoritairement reprises par les outils suivants. Par contre, il laisse de côté la classification des messages ainsi que l'inférence de la grammaire du protocole. Il ne fournit donc pas un format de message par classe de messages. Pour ce faire, les données reçues par l'application analysée sont suivies dans la trace d'exécution. Prenons l'exemple où le message est reçu dans un buffer. Avant le début de l'analyse de la trace d'exécution, chaque octet de ce buffer (qui est présent dans la mémoire de l'application) est identifié de manière unique. En effet, la trace d'exécution contient l'état de la mémoire de l'application et il est possible de lui ajouter des métadonnées telles que des identifiants. Ces octets sont dit *teintés*. *Polyglot* suit alors les manipulations de cette mémoire par les instructions de la trace, en propageant les teintes : si l'opérande d'entrée d'une instruction est teintée, alors l'opérande de sortie le sera aussi. Par exemple, si une instruction copie un octet de la mémoire teintée dans un registre, ce registre est alors teinté. Cela permet d'identifier toutes les instructions manipulant les données reçues par l'application. Cette technique appelée *DTA* (*Data Tainting Analysis*), est utilisée par la majorité des outils d'inférence applicative. Pour obtenir la sémantique des champs, *Polyglot* utilise des heuristiques sur des séquences d'instructions. Globalement, elles correspondent à la recherche dans la trace d'exécution de motifs qui reflètent cette sémantique. Les premières heuristiques les plus importantes se focalisent sur les champs `taille`. Par exemple, une trace d'exécution qui indique que la valeur d'un champ est utilisée pour délimiter un second champ laisse supposer que la valeur du premier correspond à la taille du second. La seconde étape permet de retrouver les `délimiteurs` et les `mots-clés` du message. Pour ce faire, si l'instruction de la trace est une comparaison d'un opérande teinté à une valeur fixe, alors l'identifiant de cet opérande teinté est sauvegardé dans une table nommée *table de tokens*. En analysant cette table, les `délimiteurs` et `mots-clés` sont identifiés. Par exemple, les portions successives du message comparées à une valeur fixe sont identifiées comme un seul champ délimité par cette valeur. Pour finir, ces informations sont regroupées pour former le format du message. Cette approche apporte une amélioration de la qualité de l'inférence du format des messages par rapport aux précédentes contributions, notamment celles basées sur l'identification des `mots-clés` ainsi que sur le découpage en champs du message. Cependant, tout comme dans les précédentes contributions, le format du message ne reflète pas l'encapsulation de champs. Les auteurs de *Polyglot* ont identifié une limitation importante à leur outil : si le format du message n'est pas strict, en permettant plusieurs délimiteurs par exemple (HTTP autorise l'usage des

espaces ou des tabulations comme séparateurs), alors la trace d'exécution ne correspondra qu'aux motifs d'exécution employés dans les heuristiques et cette souplesse du format ne sera pas inférée dans le format du message.

L'outil ***Rosetta*** [Caballero 2007a] est directement issu des travaux de *Polyglot* et de *Replayer*. Son but, tout comme celui de *Replayer* [Newsome 2006], est de rejouer des communications dans un autre environnement. Les auteurs étendent l'identification des champs aux messages produits en sortie par l'application en analysant les retours des appels systèmes qui sont utilisés pour construire les messages envoyés. Ils identifient également les champs de type `hash` ou `checksum` même s'ils ne dépendent pas directement de l'environnement. Enfin, ils apportent une nouvelle technique pour identifier les champs dits de `session` correspondant à une dépendance inter-message en regardant si des données d'un message reçu sont utilisées pour construire un des messages envoyés.

Trois outils d'inférence du format des messages, développés en parallèle sont issus des travaux sur *Polyglot* : *AutoFormat*, *Prospex* et *Tupni*. Le premier, ***AutoFormat***[5] [Lin 2008], repose sur l'intuition forte que différents champs d'un message sont traités par des portions de code différentes. Ainsi, le contexte d'exécution associé au traitement de chacun de ces champs est unique. Cet outil découpe donc les messages en fonction de l'ordre dans lesquels les différentes portions du code binaire sont exécutées tout en tenant compte de l'encapsulation. Par construction, le message est découpé suivant un arbre où chaque nœud représente un champ. Ainsi, des champs peuvent contenir des séquences de champs. Enfin, il retrouve les champs parallèles (séparés par un | en notation BNF) qui sont généralement traités dans des portions de code partageant un même historique de contexte d'exécution, c'est-à-dire la même pile d'appels. De par son architecture, cet outil est le seul qui n'a pas besoin d'analyser les boucles. De plus, il est le seul qui pourrait gérer une application qui traite les champs d'un message via des fonctions récursives. Il est à noter que l'on peut distinguer un nouveau défi introduit par cet outil consistant à mesurer la distance entre plusieurs portions de code.

Pour le second outil, ***Prospex*** [Wondracek 2008], Wondracek *et al.* présentent dans un premier temps des techniques d'inférence de format de message. Ainsi, ils étendent l'analyse de la *table de tokens* de *Polyglot* [Caballero 2007b] pour identifier une hiérarchie de champs. De plus, les auteurs s'appuient sur l'algorithme *Needleman & Wunsch* appliqué aux séquences de champs pour généraliser les formats obtenus. Dans un second temps, ces travaux sont poursuivis dans [Comparetti 2009] pour l'inférence de la grammaire de protocoles. La principale contribution concerne le travail sur la classification des messages à partir de plusieurs métriques. La première est issue des travaux présentés avec *Discoverer* [Cui 2007a] et concerne l'alignement de séquences de champs qui permet de définir une distance entre deux messages. La seconde métrique mesure la similarité des traces d'exécution de chacun des messages sachant que des messages similaires ont de fortes chances d'être traités par les mêmes portions du code binaire. Enfin, la dernière métrique mesure

5. Sources disponibles sur demande à l'auteur

l'impact des messages sur le système, notamment l'envoi de données sur le réseau, l'ouverture de fichiers ou la création de dossier. Ils utilisent la moyenne de ces trois valeurs comme distance entre deux messages, et regroupent les messages par classification. La grammaire du protocole est représentée sous forme d'automate fini déterministe qui est construit à partir de techniques classiques de la théorie du langage.

Peu après la publication de *AutoFormat*, Cui *et al.* ont présenté ***Tupni*** [Cui 2008]. Cet outil fait suite à leurs travaux sur *Discoverer* [Cui 2007a], où ils rencontraient des difficultés à inférer la sémantique des champs lors de la rétro-conception de format de message par une inférence réseau. Dans un premier temps, l'outil découpe le message en différents champs en fonction des instructions qui accèdent à des portions du message. Dans un second temps, les champs appartenant à une suite répétée un nombre arbitraire de fois (`*` en notation BNF) ou au moins une fois (`+` en notation BNF), sont identifiés à partir d'une analyse des boucles du programme. La troisième étape identifie les contraintes sur les champs du message, soit en utilisant les heuristiques de *Polyglot* (valeurs fixes ou champs `taille`) et *Rosetta* (dépendance inter-messages), soit à l'aide de prédicats symboliques similaires à ceux de *Replayer* (dépendances intra-messages). Enfin, *Tupni* généralise ses résultats à travers l'analyse de plusieurs messages. De plus, cet outil peut être couplé à *ShieldGen* [Cui 2007b] pour la création automatique de signatures d'attaques sur une vulnérabilité inconnue. Il peut aussi faire de l'inférence de format de fichiers en considérant ceux-ci comme des messages reçus.

C'est notamment l'approche utilisée par ***ConfigRE*** [Wang 2008] qui considère le fichier de configuration d'une application comme un message reçu par cette application. Ainsi, cet outil peut être étudié comme un outil d'inférence de format de message basée sur l'application. Son but est la rétro-conception de la configuration de contrôle d'accès utilisé par une application, par exemple la configuration des accès aux fichiers d'un serveur apache. Cet outil a des spécificités liées à son domaine d'application. Son approche pour le format de la configuration est classique : il découpe le fichier en champs puis tente d'inférer la sémantique de ces champs. La séparation en champs, tout comme l'outil *Discoverer*, dépend de délimiteurs connus. De plus, du fait du domaine du contrôle d'accès, il introduit les délimiteurs *appairés* tels que : `{` et `}` ou `(` et `)` ou de façon plus complexe `<Directory>` et `</Directory>`. Les auteurs utilisent une heuristique spécifique d'analyse des sauts conditionnels pour retrouver ces éléments. De plus, ils analysent l'interaction entre la teinte du fichier de configuration lors de son chargement en mémoire et la teinte de différentes requêtes de tests pour inférer la sémantique des champs. Cet outil utilise une approche active de l'inférence en choisissant spécifiquement les requêtes à effectuer. Ces informations permettent de construire un arbre sémantique du fichier de configuration. Enfin, il modifie les champs du fichier de configuration dont la sémantique est `permission` avec une valeur aléatoire, puis lors de l'analyse de la nouvelle trace d'exécution, identifie toutes les valeurs légitimes que peut prendre ce champs, l'application testant successivement ces valeurs légitimes contre celle de la configuration avant de sortir en échec.

Alors que les précédents outils étaient limités à l'analyse de protocoles non chiffrés, ***ReFormat*** [Wang 2009] s'attache à inférer le format de messages chiffrés. Pour cela, il doit dans un premier temps retrouver automatiquement les fonctions de chiffrement et dans la trace d'exécution, une zone de la mémoire où le message est déchiffré. Ainsi, il retrouve les fonctions possédant un grand nombre d'instructions arithmétiques qui sont généralement des fonctions cryptographiques. Dans leur outil, les auteurs utilisent la notion de *durée de vie des données* [Chow 2006] pour identifier les sorties des fonctions cryptographiques correspondant au message en clair. Enfin, ils couplent leur outil à *AutoFormat* [Lin 2008] pour faire automatiquement l'inférence du format du message. Cependant, *ReFormat* ne permet d'inférer que le format des messages reçus chiffrés par une application, pas celui des messages produits par celle-ci.

Caballero *et al.* ont poursuivi leurs travaux de *Polyglot* et *Rosetta* en incorporant des évolutions apparues dans *Tupni*, *AutoFormat*, *Prospex* et *ReFormat*. Il en découle l'outil ***Dispatcher***[6] [Caballero 2009, Caballero 2013]. Ainsi, cet outil permet de retrouver l'encapsulation des champs en utilisant une analyse plus poussée de la *table de tokens* présentée dans *Polyglot*. Les différentes sémantiques de champs inférées ont aussi été élargies. Ces travaux sont détaillés dans la thèse de J. Caballero [Caballero Bayerri 2010] de même qu'une formalisation plus poussée du format des messages inférés. Si cette formalisation est la plus poussée dans cette thèse, de notre point de vue elle souffre de la non-distinction des champs qui sont utiles à la transmission des messages (`taille, offset, hash, checksum, délimiteurs...`) de ceux qui sont utiles à l'analyse (`port, adresse IP, timestamp, nom de fichier...`). Ces derniers dépendent directement du domaine d'application de l'inférence alors que les premiers sont communs à toutes les inférences. Le but de *Dispatcher* est de permettre l'infiltration de botnets. Or, dans de nombreux cas, les malwares utilisent du chiffrement et de l'obfuscation. Ainsi, Caballero *et al.* ont étendu les techniques introduites dans *Reformat* [Wang 2009] d'identification des fonctions de chiffrement et des messages en clair au cas du bot *MegaD*. Cela leur a permis de traiter les défis suivants : multiples points de déchiffrement, déchiffrement de portions du message à la volée, identification des fonctions de chiffrement des données produites. Néanmoins, l'usage de *Dispatcher* est limité par l'absence de travail sur la classification des messages et de l'absence d'inférence de la grammaire du protocole. L'article [Caballero 2009] ne contient pas les informations sur la réalisation de ces travaux.

À notre connaissance, ***MACE*** [Cho 2010, Cho 2011] est le seul outil à pratiquer l'inférence de grammaire de protocoles par une exécution concolique (concrète et symbolique) de l'application. L'exécution concolique permet d'explorer les différentes valeurs que peuvent prendre les messages du protocole et construire sa grammaire. Dans un premier temps, les analystes devaient fournir les fonctions d'abstraction des messages d'entrée et de sortie, mais ce problème a été partiellement résolu dans [Cho 2011], où seule l'abstraction des messages de sortie est

6. Sources disponibles sur demande à l'auteur

maintenant requise. Cet outil utilise l'algorithme L^* [Angluin 1987] pour inférer la grammaire du protocole. Les auteurs ont testé leur outil sur le protocole du botnet *MegaD*, ce qui leur a permis de découvrir des portions du protocole qui n'étaient pas observables lors des phases d'analyses classiques. Si cet outil ne permet pas d'inférer le format des messages mais juste leur type, il fournit cependant des instances de messages concrètes pour arriver à un certain niveau de l'exécution. Cela est utile pour valider l'exploitabilité d'une vulnérabilité. Il a également permis de détecter des vulnérabilités dans différentes applications (trois dans Samba 3.3.4, trois dans Vino 2.26.1 et une dans RealVNC 4.1.2).

Suite aux travaux d'inférence de format du message, certaines techniques ont été utilisées pour l'inférence de structures de données, c'est notamment le cas des outils *REWARDS*, *Howard*, *ARTISTE* et *ARGOS*. Ainsi, Lin *et al.* sont les premiers à avoir adapté les techniques d'inférence de format de message à l'inférence de structure de données dans l'outil ***REWARDS*** [Lin 2010, Lin 2011]. Il permet de pratiquer des *analyses forensics de la mémoire* ainsi que du fuzzing. Pour ce faire, ils utilisent la même analyse de trace d'exécution que dans *Dispatcher* [Caballero 2009] pour retrouver le format des structures de données (correspondant à des messages pour *Dispatcher*) et la sémantique des champs. Cependant, ils doivent aussi pouvoir analyser la mémoire à un moment spécifique de l'exécution. Ainsi, ils étendent leur analyse en ajoutant une analyse en arrière de la trace d'exécution pour retrouver d'où proviennent les éléments en mémoire.

Le second outil, ***Howard*** [Slowinska 2010, Slowinska 2011], a été conçu pour aider le débogage d'applications binaires. Il enregistre les buffers alloués dynamiquement, puis identifie la manière dont ils sont accédés pour déterminer leur découpage en différents champs. Cette étape est similaire aux travaux dans *FFE/x86* [Lim 2006] ou *Tupni* [Cui 2008]. Cependant, son analyse dynamique lui permet une plus grande précision des résultats. En effet, il n'y a pas de surapproximation des valeurs prises par les champs de la structure. De plus, il inclut une amélioration de l'inférence de tableaux présentée dans *REWARDS* et *Polyglot* [Caballero 2007b] et il propage de manière complémentaire à *REWARDS* les structures issues de fonctions connues. Pour finir, en plus d'aider au débogage d'application binaire, *Howard* peut détecter et empêcher des corruptions en mémoire en instrumentant l'application d'origine, parfois de manière manuelle, pour sécuriser les déréférencements de pointeurs.

Peu après, en collaboration avec les auteurs de *REWARDS*, Caballero *et al.* ont développé ***ARTISTE*** [Caballero 2012]. Cet outil étend des techniques présentées dans *Dispatcher* [Caballero 2009] et dans *REWARDS* [Lin 2011] pour l'inférence de structures et de format de messages. Ainsi, il peut inférer des structures complexes telles que des arbres ou des listes doublement chaînées. Il présente également une nouvelle technique permettant d'identifier des structures similaires allouées à différents endroits du programme. Enfin, il améliore la vitesse d'inférence des structures simples.

Enfin, Zeng et Lin ont développé ***ARGOS*** [Zeng 2015] pour aider à la compréhension et au monitoring des noyaux des systèmes d'exploitation. Ils utilisent les

précédentes techniques présentées dans *ARTISTE* pour propager l'inférence de sémantique sur les structures du noyau. De plus, ils utilisent le contexte d'exécution, c'est-à-dire les actions faites par le noyau telles que la création de processus ou la lecture de fichiers, pour classifier les différents objets du noyau, et leur donner un sens compréhensible par un humain.

Comme présenté en section 1.2, les fuzzers intelligents utilisent un modèle de format de message pour pratiquer des tests en profondeur. Dans ce contexte, ***Fuzzgrind***[7] [Campana 2009b, Campana 2009a] essaie d'automatiser l'inférence d'un modèle de format de messages en exécutant symboliquement l'application. Il utilise le framework *Valgrind* [Nethercote 2007] comme cœur de son exécution symbolique, ce qui lui permet d'enregistrer les différents chemins d'exécution de l'application. Ils associent à chacun de ces chemins d'exécution une classe de message. Ainsi, les différentes contraintes sur les données d'entrées reflètent le format de ces classes de message associées.

L'outil de fuzzing ***AFL***[8] [Zalewski 2017] lui aussi infère des modèles en fonction de l'exploration d'une application binaire. Cependant, cette exploration est basée cette fois sur une exécution concrète de l'application couplée avec un algorithme de mutation des entrées. La qualité des mutations est directement déterminée par le chemin d'exécution empreinté. Cet outil est utilisé par une large communauté d'analystes et a participé à la découverte de nombreuses vulnérabilités.

TABLE 1.4 – Résumé des outils d'inférence de structures. Un tiret indique que l'outil ne traite pas cet aspect de l'inférence

Outils	**Méthode d'analyse**	**Classification des structures** *Mesure basée sur :*	**Inférence de la structure**	
			Type de structure	*Généralisation du modèle utilisé*
REWARDS	Passif dynamique	Contexte d'exécution	Simple	Non
Howard	Passif dynamique	-	Simple	Non
ARTISTE	Passif dynamique	Contexte d'exécution	Récursive	Oui
ARGOS	Passif dynamique	-	Récursive	Non
Fuzzgrind	Actif symbolique	Chemin d'exécution	simple	Non
AFL	Actif dynamique	Chemin d'exécution	Récursive	Non

7. Code source disponible à `https://github.com/Grindland/Fuzzgrind`
8. Code source disponible à `http://lcamtuf.coredump.cx/afl/`

Synthèse

TABLE 1.5 – Résumé des outils d'inférence applicative. Un tiret indique que l'outil ne traite pas cet aspect de l'inférence de protocole

Outils	**Méthode d'analyse**	**Classification des messages** *Mesure basée sur :*	**Format du message** *Reçus Envoyés*	*Modèle utilisé*	*Généralisation du modele*	**Grammaire du protocole** *Modèle utilisé*
FFE/x86	Statique	-	Envoyés	Expression régulière	Oui	-
Replayer	Active dynamique	-	Reçus Envoyés	Template	Non	-
Polyglot	Passive dynamique	-	Reçus	Template annoté	Non	-
Rosetta	Passive dynamique	-	Reçus Envoyés	Template annoté	Non	-
AutoFormat	Passive dynamique	-	Reçus	Arbre annoté	Oui	-
Prospex	Passive dynamique	Alignement + comportement de l'application	Reçus	Arbre annoté	Oui	Automate fini déterministe
Tupni	Passive dynamique	-	Reçus	Arbre annoté	Oui	-
ConfigRE	Active dynamique	-	Reçus	Arbre annoté	Oui	-
ReFormat	Passive dynamique	-	Reçus	-	-	-
Dispatcher	Passive dynamique	-	Reçus Envoyés	Arbre annoté	Non	-
MACE	Active concolique	Abstraction manuelle des messages produits	-	-	-	Machines de Mealy

Le tableau 1.5 résume les différentes approches d'inférence de protocole basée sur une analyse d'une application et le tableau 1.4 identifie les outils pratiquant l'inférence de structures de données. L'approche statique ou dynamique et active ou passive est résumée en seconde colonne. Pour le tableau 1.5, la troisième colonne met en évidence les outils pratiquant le regroupement de messages pour inférer la grammaire de protocole (dernière colonne). Ces modèles sont classés suivant leur niveau de représentativité, rappelons que ces modèles sont brièvement présentés en section 1.3.3. Presque tous ces outils infèrent le format des messages reçus. Cependant, peu d'entre eux sont capables d'inférer le format des messages produits par une application. La colonne `généralisation du modèle` permet d'identifier les outils capables de généraliser le format de message, soit en obtenant directement un format générique (*FFE/x86*), soit en analysant plusieurs messages. Pour les structures de données (tableau 1.4) le type de structures identifiée est important. Ainsi, deux types de structures sont identifiés, les structures simples (références directes) et les structures récursives (listes chainées, arbres...). Ces tableaux mettent en évidence que *FFE/X86* est le seul à employer une analyse statique du binaire, alors que les autres outils utilisent une approche dynamique, dont la majorité de manière passive. De plus, les outils s'intéressent principalement au format du message et négligent la classification de ces messages, ce qui constitue une limite à l'inférence de la grammaire du protocole. En conséquence, seuls *Prospex* et *MACE* qui classent les messages, permettent d'inférer la grammaire du protocole. Nous constatons également une ouverture des outils vers des perspectives nouvelles avec leur adaptation à l'inférence de format de fichiers (*Tupni* et *ConfigRE*) et à l'inférence de structures de données, notamment pour le noyau (*REWARDS*, *Howard*, *ARTISTE* et *ARGOS*). Les tableaux 1.6 et 1.7 résument les contributions majeures de chacun des outils ainsi que le défi de la rétro-conception de protocole auquel il répond.

1.5 Conclusion

Dans ce chapitre dédié à la rétro-conception de protocole, nous avons présenté les différentes méthodes et étapes de la rétro-conception. De plus, nous avons mis l'accent sur les étapes préliminaires à l'inférence de protocoles. Ces étapes sont partiellement abordées par les outils et nécessitent encore souvent une intervention directe de l'analyste. Si cette intervention manuelle est encore nécessaire aujourd'hui, c'est que ces différentes étapes contiennent elles-mêmes un certain nombre de défis, présentés dans la section 1.3, qui ne sont pas encore résolus entièrement, et donc pas nécessairement abordés par les outils de rétro-conception. Ces défis constituent un des freins majeurs à l'obtention d'une méthodologie de rétro-conception générique et efficace.

Aussi, comme le soulignent Collberg *et al.* [Collberg 1997], ces faiblesses sont à considérer pour développer des techniques résiliantes à la rétro-conception. C'est précisément ce que nous proposons de faire dans ces travaux de thèse, dans le domaine de l'obfuscation. Notre contribution est de profiter des faiblesses et limita-

TABLE 1.6 – Résumé des contributions majeures de chaque outil d'inférence applicative et du challenge qui est résolu par cette contribution

Outil	Contribution	Défi résolu
FFE/x86	Analyse statique du binaire de l'application à partir d'un point de synchronisation	Inférence du format des données générées par une application sans exécution
RePlayer	Rejeu de traces par obtention des contraintes liées à l'environnement	Rejeu de traces dans un autre environnement
Polyglot	Analyse de la trace d'exécution d'une application pour inférer le format des messages reçus	Inférence du format des messages reçus par l'application
Rosetta	Extension des travaux de *Polyglot* aux messages produits : analyse des retours d'appels système	Inférence des messages produits par l'application
AutoFormat	Découpage en champs basé sur la région d'exécution du code	Traitement d'un message par une fonction récursive + nouvelle technique de découpage de message en champs
Prospex	Généralisation par Needleman & Wunsch du format de message + analyse du système servant de métrique pour le clustering de messages	Généralisation du format de message + nouvelles métriques pour le clustering
Tupni	Intégration des travaux précédents + généralisation par analyse de multiples messages	Généralisation du format des messages
ConfigRE	Analyse dynamique active du traitement du fichier de configuration	Inférence de délimiteurs appairés dans un fichier de configuration
ReFormat	Identification automatique des interfaces de chiffrement par analyse statistique des instructions arithmétiques	Inférence de messages chiffrés
Dispatcher	Extension *ReFormat* aux messages générés + formalisation du modèle de format de message	Défi concret d'un botnet : multiples interfaces de chiffrement, pas d'accès au serveur de contrôle
MACE	Inférence concolique	Exploration automatique des chemins de traitement des messages reçus par l'application

TABLE 1.7 – Résumé des contributions majeures de chaque outil d'inférence de structures de données et du challenge qui est résolu par cette contribution

Outil	Contribution	Défi résolu
REWARDS	Inférence de structures de données par analyse montante à partir des arguments d'appels système prototypés par l'utilisateur	Analyse forensic d'une trace d'exécution
Howard	Typage automatique du format des structures simples allouées dynamiquement	Typage automatique des structures allouées dynamiquement
ARTISTE	Analyse des déréférencements de pointeurs dans une trace d'exécution	Inférence des structures de données complexes : listes chainées, arbres, tableaux
ARGOS	Analyse des actions du système pour la classification automatique des structures allouées dynamiquement dans le noyau Linux	Aider un humain à comprendre et analyser les structures du noyau et comment elles sont accédées
Fuzzgrind	Exécution symbolique et application des contraintes sur les données d'entrée	Génération automatique d'un modèle pour les fuzzer intelligents
AFL	Exécution concrète et algorithme génétique de mutation des entrées	Génération automatique de "tests-case" pour l'exploration de chemin d'exécution d'application

tions actuelles de la rétro-conception pour définir une nouvelle approche de l'obfuscation de protocoles, dans l'objectif de rendre la rétro-conception de ces protocoles particulièrement complexe. Le prochain chapitre est donc dédié à une étude de l'obfuscation logicielle de protocoles ainsi que des protections existantes et le chapitre 3 est consacré à la présentation de notre approche d'obfuscation de protocoles.

Chapitre 2

Obfuscations de logiciels et de protocoles

Sommaire

Les techniques d'obfuscation ont été développées afin de protéger les logiciels. L'obfuscation a deux grandes applications :

— la protection d'une propriété intellectuelle contre sa rétro-conception, on parle alors de **confusion** ;
— la **dissimulation** de la présence d'une donnée ou fonctionnalité.

Si on se place du point de vue d'une entreprise légitime, alors cette entreprise peut chercher à protéger la connaissance d'éléments critiques d'une application (tels que les détails d'implémentation d'un algorithme). Ces éléments pourraient être retrouvés par rétro-conception et être utilisés à des fins malveillantes, entrainant des pertes financières ou matérielles. Les obfuscations de ce type cherchent généralement à complexifier la compréhension globale d'un élément critique.

D'un autre côté, le développeur d'une application malveillante va souvent chercher à ce que son code malveillant soit difficile à détecter. Pour cela, il peut utiliser des obfuscations visant à complexifier la détection de son produit par les sondes anti-virus ou réseaux. Une grande variété de *packers* [Roundy 2013] ont été développés à cette fin.

Aujourd'hui, les produits logiciels déployés sont de plus en plus complexes, ils comportent divers composants qui peuvent communiquer entre eux via des protocoles. Les obfuscations de protocoles proposées jusqu'alors visent principalement à

cacher les communications aux différentes sondes qui pourraient être mises en place. À notre connaissance, il n'y a pas de travaux qui prennent en compte explicitement les défis auxquels sont confrontés les outils de rétro-conception de protocoles tels que nous les avons discutés au chapitre 1, pour rendre la rétro-conception de protocoles plus complexe. Ainsi, nous proposons dans cette thèse une nouvelle cible pour l'obfuscation, au sens de la protection contre la rétro-conception : les protocoles de communication. Nous proposons également une approche réalisant une telle obfuscation aux chapitres 3 et suivants. Ce chapitre présente quant à lui un état de l'art de l'obfuscation et les principales définitions associées.

2.1 Terminologie

Historiquement l'obfuscation s'intéresse à la protection des programmes contre la rétro-conception. L'obfuscation au sens de la dissimulation d'un flux réseau n'apparaît que plus tard et est moins étudiée. Dans cette section nous reprenons la terminologie développée dans le cadre de la protection logicielle.

La définition d'obfuscation communément admise est celle introduite par Collberg *et al.* dans [Collberg 1997].

Définition 1 *Une obfuscation O transforme un programme P en un programme $O(P)$ ayant le même comportement observable que P mais dont la rétro-conception est plus complexe. Il doit de plus respecter les conditions suivantes :*

- *Si le programme P ne termine pas ou termine en erreur, alors le comportement de $O(P)$ n'est pas défini ;*
- *Si le programme P termine, alors le programme $O(P)$ doit terminer et produire les mêmes sorties que P.*

La notion de *comportement observable* est un élément important de cette définition, nous y reviendrons plus tard à travers les contributions de Wang Chexi [Wang 2001] et Dalla Preda [Dalla Preda 2007].

[Collberg 1997] identifie quatre cibles pour les obfuscations :

- L'*organisation* du code source. Ce type d'obfuscation présente un intérêt limité dans le cas de codes compilés : une partie des informations relatives au code source n'est déjà plus présente ;
- Les *données*, qu'elles soient échangées, stockées ou manipulées en mémoire ;
- La *structure de contrôle* du code, i.e. son organisation algorithmique ;
- Les *challenges* de la rétro-conception.

Notons que dans cette définition, les données échangées sont donc déjà vues comme une cible potentielle de l'obfuscation.

L'évaluation de la qualité d'une obfuscation est encore aujourd'hui difficile. Collberg *et al.* proposent quatre axes pour évaluer une obfuscation :

- Sa *puissance*, i.e., à quel point le programme est plus complexe à comprendre pour un humain ;
- Sa *résistance* face à des outils d'analyse automatique ;

— Son *coût* en terme de mise en place, de surcharge en temps d'exécution ou de mémoire utilisée ;
— Sa *furtivité*, *i.e.*, à quel point il est facile d'identifier les parties obfusquées de celles qui ne le sont pas.

Ils suggèrent également des mesures de complexité de code telles que la taille du programme, la complexité cyclomatique, la complexité des structures et d'autres mesures classiques de qualité de code.

Ceccato *et al.* ont mené une étude empirique sur 44 différentes obfuscations (sur le code source Java) pour voir leurs effets sur 10 différentes mesures de complexité de code [Ceccato 2015]. Ils ont ainsi pu montrer que certaines obfuscations n'ont d'impact que sur une partie des métriques. De plus, ils ont pu mettre en exergue une dépendance entre la complexité initiale du code et la complexité du code obfusqué : plus une application est complexe au départ, et plus elle sera facile à obfusquer. En effet, une application plus complexe a plus d'éléments sur lesquels travailler et il n'est pas nécessaire de créer artificiellement de la complexité.

Cette étude a permis de caractériser la *puissance* des obfuscations. Par ailleurs, Banescu *et al.* [Banescu 2017] se sont intéressés à mesurer et prédire la *résistance* des obfuscations face à des outils d'analyse automatique, plus particulièrement aux outils de rétro-conception basés sur une exécution symbolique du programme. En effet, les outils d'exécution symbolique permettent de construire les contraintes sur les données ou les branchements conditionnels au fur et à mesure de l'exécution symbolique. Cependant, dans le cas de prédicats opaques, le but est aussi de contourner ces analyse en rendant le calcul non réalisable. Pour pallier à ces difficultés, des outils d'exécution dynamique symbolique tels que *BinSec* [David 2016] permettent une exécution concrète de certaines parties du programme, notamment les prédicats opaques. L'avantage de *BinSec* pour de tels cas est la forte interaction entre l'exécution concrète et l'exécution symbolique.

Revenons à présent sur la notion de comportement observable introduite dans la définition de l'obfuscation par Collberg *et al.*. Wang Chenxi, dans [Wang 2001], précise cette notion. Plutôt que de caractériser un comportement observable par les interfaces du programme obfusqué avec ses bibliothèques, il se place à un niveau plus abstrait : les fonctionnalités du programme. Par exemple, si l'on considère une fonctionnalité d'écriture dans un fichier alors il est possible d'utiliser plusieurs séquences de fonctions différentes (par exemple, pour le langage C sous Linux) :

— fopen, fwrite, fclose
— open, send/write, close

Si l'on considère comme comportement observable les appels à la libc, le remplacement de la première séquence d'appels par la seconde est une obfuscation qui n'est pas permise, car elle change le comportement observable. Si on se place au niveau de la fonctionnalité, l'obfuscation est valide car la même écriture est réalisée dans le même fichier.

Dalla Preda définit plus formellement cette notion de fonctionnalité ainsi qu'un modèle de l'attaquant dans [Dalla Preda 2007] se basant sur l'interprétation abstraite. L'interprétation abstraite permet d'étudier la sémantique des programmes

et ses approximations. Au niveau le plus précis, ou concret, la sémantique d'un programme est l'ensemble des traces d'exécutions possibles. Chaque trace étant la séquence précise des opérations effectuées et de l'état du système. On parle de sémantique de traces. La fonctionnalité à préserver est une sémantique plus abstraite. En général on se base sur la relation entrée/sortie, qui est une abstraction de la sémantique de traces : pour chaque trace, on ne conserve que l'état des entrées au début de la trace et l'état des sorties à la fin de la trace. Dalla Preda s'appuie sur une hiérarchie des sémantiques abstraites pour caractériser les obfuscations : une obfuscation est mise en relation avec la sémantique la plus concrète qu'elle préserve. Les obfuscations les moins puissantes ne modifient pas la sémantique de trace, les plus puissantes ne conservent que la sémantique utilisée pour définir les fonctionnalités à préserver. L'attaquant, quant à lui, est vu comme un interpréteur abstrait. Il est capable d'effectuer des calculs dans un certain domaine abstrait, avec une certaine précision. L'objectif de l'attaquant est de construire une sémantique du programme obfusqué aussi précise que possible à son niveau d'abstraction.

Sans entrer dans le formalisme de l'interprétation abstraite, la flexibilité que nous apporte cette notion de comportement observable nous permet de considérer plusieurs programmes communiquant ensemble comme une seule entité. Les canaux de communication n'étant plus observables, c'est-à-dire non pertinents du point de vu des utilisateurs, nous pouvons nous permettre d'effectuer des obfuscations qui ne préservent pas les messages échangés.

Ce type d'obfuscation a déjà été proposé, comme nous le verrons plus loin, l'objectif étant de dissimuler l'existence de certains flux à un observateur sur le réseau. Cependant, ces obfuscations ne résistent pas à un attaquant dont l'objectif serait de comprendre le protocole de communication, pour interagir avec le système par exemple. Les obfuscations que nous proposons dans le cadre de cette thèse ciblent spécifiquement ce type d'attaquant. C'est pourquoi nous introduisons les notion de **confusion** et **dissimulation** pour distinguer deux objectifs possibles de l'obfuscation : ralentir la compréhension, ou cacher l'existence d'une fonctionnalité ou information. Dans la suite de ce chapitre, nous présentons les obfuscations sur les logiciels puis les obfuscations sur les protocoles.

2.2 Obfuscation logicielle

L'obfuscation logicielle peut s'appliquer à différentes étapes du processus de développement logiciel. Ce processus commence par la spécification de l'architecture. Dans le cadre d'une application distribuée, qui sauvegarde des données en dehors de l'application, un exemple d'architecture peut être donné par la Figure 2.1. Une architecture bien pensée permettra d'identifier les systèmes critiques à protéger. La seconde étape est le développement à proprement parler, avec la compilation du code source. De nombreuses transformations sur le code source existent pour rendre celui-ci plus complexe. Ces transformations utilisent le même processus d'analyse syntaxique et d'abstraction du code source que les compilateurs afin de faciliter l'im-

plémentation de ces transformations. Elles sont appelées transformations *pre-build*. Si l'application est compilée, certaines transformations peuvent intervenir directement sur l'application binaire générée, indépendamment du code source d'origine. Ce sont des transformations *post-build*. Ensuite, dans un processus de développement classique, intervient le test. Il ne devrait pas y avoir d'obfuscation à ce niveau, mais les obfuscations doivent être prises en compte durant les diverses campagnes de tests, notamment pour vérifier que l'application obfusquée reste conforme à sa spécification. Enfin, l'application est déployée.

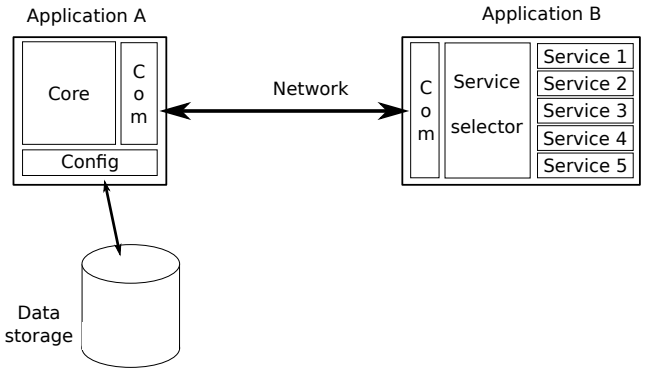

FIGURE 2.1 – Application distribuée avec système de stockage des données externe.

2.2.1 Obfuscations appliquées sur le code source : *pre-build*

Dans un premier temps, les obfuscations s'appliquaient principalement sur le code source. Dans le cas d'applications compilées vers du code assembleur, elles ciblaient l'étape de décompilation pour la rendre plus complexe. Collberg *et al.* ont développé de nombreuses techniques d'obfuscations pour des applications Java dans [Collberg 1998a, Collberg 2003]. Ces techniques s'appliquent sur différents points du développement de programmes et sont en partie spécifiques au langage Java qui garde des traces du code source dans le code compilé afin notamment de permettre la réflexivité du code. Ainsi, une partie des obfuscations proposées sont les mêmes que celles qui seraient proposées pour un langage interprété. C'est aussi le cas du langage C++ sous Windows avec le "*Run-Time Type Information* (*RTTI*) [Microsoft 2017]. Dans [Collberg 2007], les auteurs développent des transformations qui s'appliquent directement sur le bytecode Java (code compilé Java). Pour appliquer ces transformations, ils utilisent les mêmes abstractions et parseurs de code que le compilateur utilise par la suite. Cependant, de plus en plus d'outils et d'analyses de rétro-conception utilisent directement les applications binaires sans avoir besoin de faire de la décompilation. Pour faire face à cela, de nouvelles obfuscations ont été créées pour s'appliquer sur le binaire. De plus, ces obfuscations ont elles aussi pour conséquences de rendre la décompilation plus complexe.

Modification de la mise en forme : Cette obfuscation s'applique directement sur le code source. Elle est surtout utile pour les langages interprétés. Elle modifie la structure visuelle du code sans changer son interprétation par le parseur de code qui le compilera ou fera son exécution. Cette obfuscation est de type *confusion*. Des exemples sont disponibles sur le site `https://www.cise.ufl.edu/~manuel/obfuscate/obfuscate.html`.

Renommage du code : Cette obfuscation concerne le nommage des fonctions ou des variables. Le nom d'origine qui a une signification dans la plupart des cas est changé par un nom abstrait, par exemple une suite aléatoire de lettres. Cette obfuscation entre dans la catégorie *confusion*.

Modification du graphe de flot de contrôle : Cette obfuscation modifie le flot de contrôle d'origine. Plusieurs solutions existent. Collberg *et al.* ont introduit l'utilisation de prédicats opaques [Collberg 1998b] qui permettent de masquer la cible d'un branchement de code. Cette solution a évolué vers l'applatissement des appels : tous les appels passent par une seule et même fonction qui effectue la redirection. Enfin, le graphe de flot de contrôle peut être modifié par ce qui est communément appelé du *code spaghetti*. De nombreux branchements de code (ne modifiant pas la fonctionnalité du code) sont ajoutés au graphe d'origine afin de masquer les branchements importants dans du bruit. Le classement de ces obfuscations en *dissimulation* ou en *confusion* depend surtout de la façon dont elles sont utilisées. Dans l'exemple 1 ci-après, les obfuscations sont utilisées pour masquer l'accès à un algorithme critique. Elles sont alors considérées comme faisant partie de la classe *dissimulation*. Dans le second cas (exemple 2), les obfuscations sont utilisées directement sur l'algorithme pour le rendre plus complexe à comprendre, elles seront alors donc qualifiées de *confusion*.

Exemple 1 *Dans cet exemple, l'obfuscation tente de dissimuler l'appel à l'algorithme critique. L'état initial du graphe d'appel est donné par la* Figure *2.2. Un premier traitement est effectué dans le bloc de code 8, puis un second traitement est effectué dans le bloc de code 9, avant d'entrer effectivement dans l'algorithme. En cas d'erreur le programme termine dans l'un des blocs de retour en erreur. La* Figure *2.3 est obtenue en ajoutant des liaisons supplémentaires au graphe d'appel. Ensuite, tous les appels sont obfusqués par le passage par une fonction de prédiction de branchement par prédicats opaques. A la fin de chaque bloc, un prédicat est donné à la fonction de branchement pour savoir où aller dans le code ensuite. Cela est illustré par la* Figure *2.4. Ainsi, la localisation et même potentiellement l'existence de l'algorithme critique est dissimulée.*

Exemple 2 *Dans cet exemple, les mêmes transformations peuvent être appliquées mais sur le graphe de flot de contrôle de l'algorithme même. Ainsi la compréhension du fonctionnement de l'algorithme est plus complexe. Ces transformations seront dites de confusion. Les même figures 2.2, 2.3 et 2.4 peuvent être utilisées pour*

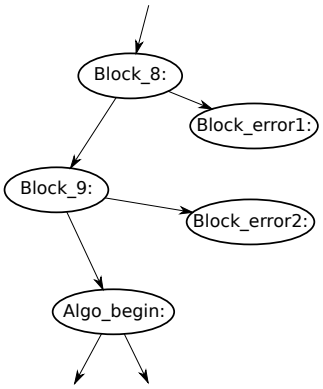

FIGURE 2.2 – Graphe d'appel vers un algorithme critique avant obfuscation.

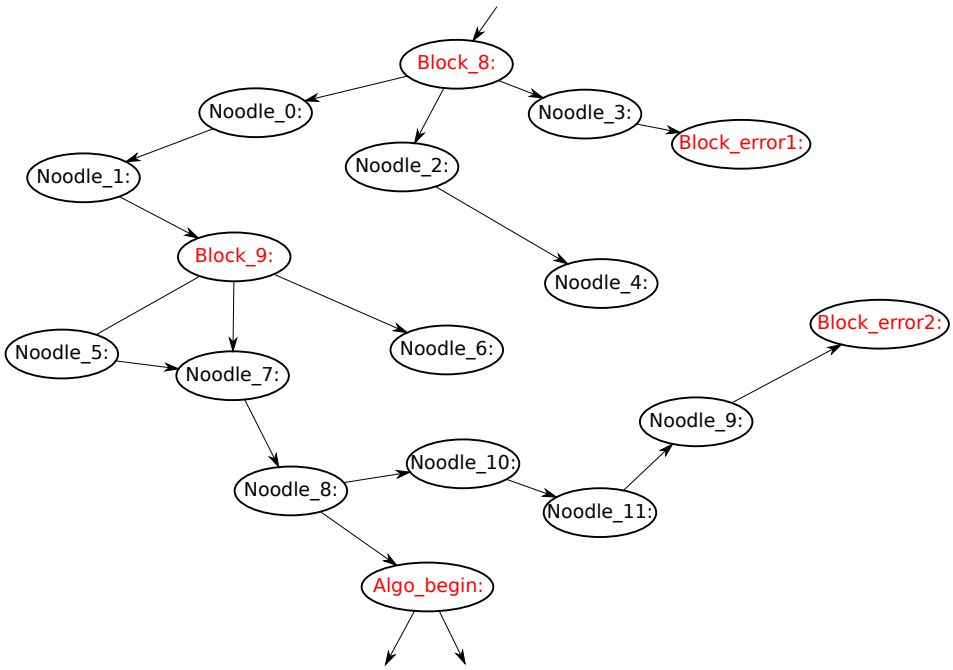

FIGURE 2.3 – Graphe d'appel vers un algorithme critique après obfuscation par insertion de code spaghetti.

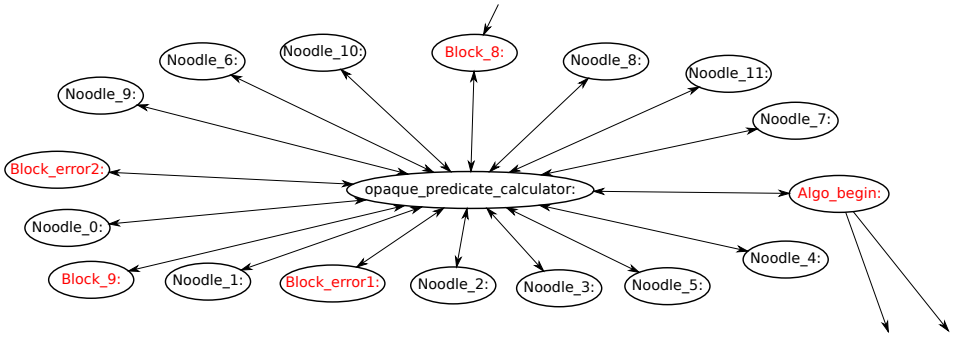

FIGURE 2.4 – Graphe d'appel vers un algorithme critique après obfuscation par insertion de code spaghetti puis aplatissement du graphe d'appel par l'utilisation de prédicats opaques.

illustrer cet exemple si l'on considère que le nœud d'entrée est Algo_begin et non plus Block_8.

2.2.2 Obfuscations appliquées sur le binaire : *post-build*

Les modèles d'obfuscation sont spécialisés en fonction du domaine auquel ils s'appliquent. Ainsi, dans [Wroblewski 2002] une application est modélisée par une suite d'instructions, ce qui permet de définir les transformations d'obfuscations comme des modifications de suites d'instructions assembleur au niveau du binaire. Dans un premier temps, des suites équivalentes d'instructions assembleur permettent de créer de la diversification automatique dans le binaire. Dans un second temps, l'obtention d'un graphe de flot de contrôle cohérent a été complexifiée en proposant des prédicats opaques pour résoudre les branchements du code assembleur. Un algorithme permettant d'appliquer les obfuscations en dernière étape, post-build, a été proposé. L'hypothèse considérée dans ce modèle est que l'attaquant effectue des analyses statiques du code, ainsi ces obfuscations ont une très bonne *puissance* et une très bonne *résistance*. Cependant, ces obfuscations ont une *furtivité* très faible ce qui permet à un analyste d'effectuer une exécution concrète des prédicats opaques afin de résoudre leur destination, diminuant ainsi la *résistance* globale des obfuscations. En parallèle, d'autres obfuscations binaires ont été proposées, principalement pour modifier le graphe de flot de contrôle [Cho 2001, Linn 2003, Wang 2000]. Là encore, ces obfuscations ont visé l'analyse statique de binaire.

Par la suite, dans [Cappaert 2010, Schrittwieser 2011], des techniques visant aussi les outils d'analyse dynamique ont été proposés. Blazy *et al.* ont proposé des techniques qui ciblaient spécifiquement les outils dynamiques afin de fausser leurs résultats tout en augmentant la crédibilité de ces résultats *i.e.*, la *furtivité* de leurs obfuscations. Toutes ces obfuscations sont de type *confusion*, mais si leur utilisation est intégrée à un *packer* de code [Collberg 2007], alors la compréhension finale du code n'est pas modifiée, seul son accès est masqué et alors il s'agit d'obfuscations

de type *dissimulation*. De plus, les obfuscations qui utilisent des exécutions équivalentes [Dalla Preda 2007] sont aussi à classer en *dissimulation* car la compréhension n'est pas modifiée mais le chemin d'exécution va essayer de contourner les appels qui sont surveillés par les anti-virus.

Durant ces dernières années, de nouvelles techniques d'obfuscation à base de virtualisation sont apparues. Ces techniques sont implémentées dans divers outils du commerce tels que *Agile.NET*[1], *Crypto Obfuscator*[2] ou *Eazfuscator.NET*[3] pour le code source en `C#`, ou des outils tels que *VmProtect*[4] ou *Tigress*[5] [Banescu 2016] pour les code source en C. Le code compilé, *i.e.* les instructions assembleurs sont virtualisées dans une machine virtuelle, ainsi seul le code de la machine virtuelle reste directement accessible, le code à protéger est lui réencodé dans un bytecode qui pourra s'exécuter dans cette machine virtuelle. Cela complexifie grandement l'analyse statique du code car cela implique souvent d'écrire un désassembleur pour cette machine virtuelle spécifique. De plus, pour des outils tels que *VmProtect* ou *Tigress* les machines virtuelles (VM) sont différentes à chaque compilation (changement du bytecode, du code de chaque instruction virtuelle, mode d'enchainement des opcodes du bytecode...), ce qui implique que pour chaque nouveau binaire obfusqué, l'analyse de la VM doit être effectué à nouveau. Cependant, [Salwan 2018] a proposé une méthode visant à automatiser cette analyse de la VM *Tigress* et désobfusquer le code pour réobtenir le code d'origine. Les travaux de *Tigress* [Banescu 2016] sont étendus dans [Kuang 2018]. Ils implémentent un schéduleur pour avoir une exécution du bytecode non linéaire.

2.2.3 Obfuscations appliquées sur les données

Ces dernières années, des mécanismes de protection ont été imaginés pour les données, principalement au niveau de la manipulation des données en mémoire. Les transformations proposées dans [Lin 2009] ont pour but de modifier les structures mémoire manipulées par une application. À chaque nouvelle compilation, une nouvelle instance de structures est créée. Ainsi, la compréhension d'une instance des structures ne permet pas directement la compréhension d'une nouvelle instance déployée. Cette obfuscation est qualifiée de *confusion*. De plus, si une attaque utilise des structures qui ont été obfusquées, alors, cette attaque est valide sur cette instance spécifique et ne l'est probablement plus sur une autre instance obfusquée différemment de cette même application. Cette technique a pour conséquence d'introduire du polymorphisme dans les différentes versions déployées d'une application. Cette manipulation des données était déjà proposée par Collberg *et al.* dans [Collberg 1998a].

Pour les données qui sont stockées en dehors de l'application, la protection la plus courante est le chiffrement. Il s'agit d'obfuscation de type *dissimulation* car on perd

1. `https://secureteam.net/acode-features-detailed`
2. `https://www.ssware.com/cryptoobfuscator/obfuscator-net.htm`
3. `http://www.gapotchenko.com/eazfuscator.net`
4. `http://vmpsoft.com/support/user-manual/introduction/what-is-vmprotect/`
5. `http://tigress.cs.arizona.edu/`

l'accès aux données après le chiffrement. Mais si le chiffrement est cassé (exemple de la faille de sécurité "*heartbleed*" permettant de retrouver les clés de chiffrement), la compréhension des données et la façon dont elles sont manipulées par l'application n'est pas affectée. La figure 2.5 montre une application qui sérialise (dans le module A) et envoie de données qui sont protégées par obfuscations sur le message, puis, par chiffrement. Cependant, comme le montre cette figure, il existe un point du programme (la loupe), correspondant à l'appel de la fonction d'obfuscation, point de passage obligatoire pour la protection des données, où l'on peut accéder à la donnée en clair quelles que soient les obfuscations effectuées après cette interface. La librairie de chiffrement (ou d'obfuscation) peut être obfusquée au sens logiciel si le résultat du chiffrement (ou de l'obfuscation du message) n'est pas modifié.

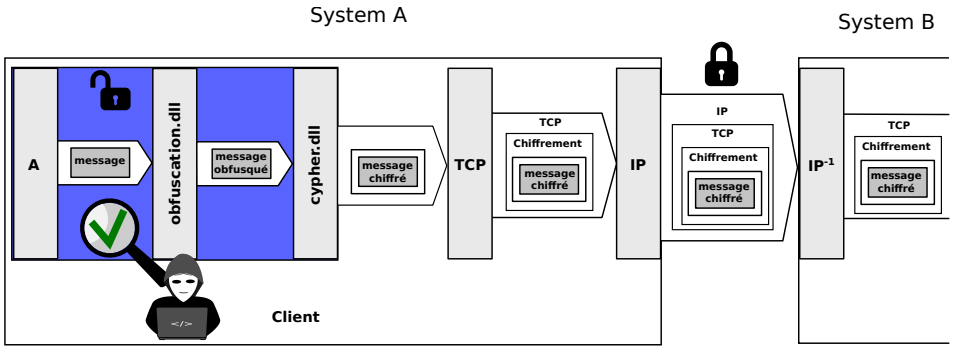

FIGURE 2.5 – Exemple de code utilisant une obfuscation de protocole classique couplée avec de la cryptographie. Un attaquant a accès dans l'application au buffer en clair.

Pour protéger ce goulet d'étranglement, il faut que les données soient directement sérialisées obfusquées avant l'appel à la fonction de chiffrement. Ainsi, une solution est d'appliquer des obfuscations directement sur la sérialisation des données. En conséquence, ces obfuscations doivent être elles aussi partagées entre le client et le serveur pour garder une cohérence entre la sérialisation des données d'un côté et le parsing des données de l'autre. Cela est illustré par la figure 2.6. Les données sont tout d'abord sauvegardées dans une structure interne opaque via des accesseurs. Les données sont ensuite directement sérialisées depuis cette structure opaque vers un message obfusqué. La fonction de sérialisation contient donc les obfuscations à appliquer. Un attaquant ne peut pas retrouver facilement (pas de présence de message en clair à l'interface de la fonction de sérialisation) le message d'origine. C'est la solution qui a été adoptée dans cette thèse et qui est présentée dans les chapitres 3 et 4. La section suivante présente les protections de données existantes dans le cadre spécifique des protocoles de communication.

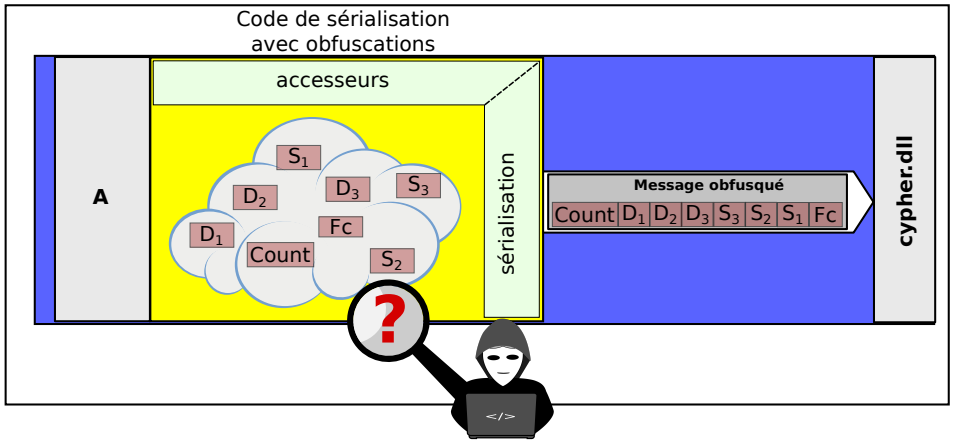

FIGURE 2.6 – Exemple de code où la sérialisation des données contient les obfuscations. Les données sont stockées dans une structure opaque via des accesseurs spécifiques. Il n'y a pas de point du programme où le buffer est en clair.

2.3 Obfuscation de protocoles

Une des particularités de l'obfuscation de protocoles est que non seulement le protocole en lui même peut être obfusqué, mais également son canal de communication ainsi que l'application qui le traite. Dans la pratique, les protections de protocoles existantes correspondent principalement aux transformations sur le stockage et l'encodage des données du modèle de Collberg *et al.* [Collberg 1997], i.e., des obfuscations sur le canal de communication. Cela correspond dans notre cas à des obfuscations de type *dissimulation*. L'exemple le plus courant est l'utilisation du chiffrement. [Dyer 2015] propose une classification plus détaillée des techniques d'obfuscation de protocoles :

— la *randomisation* : les données échangées ne sont pas identifiables ;
— l'*imitation* : les communications ressemblent à des communications légitimes ;
— le *tunneling* : les données sont encapsulées dans des communications légitimes ;
— les *systèmes programmables* : qui combinent les avantages de la randomisation et de l'imitation.

2.3.1 Randomisation

Le but de la randomisation est de faire en sorte que la communication ne ressemble à rien d'identifiable. Cette transformation doit enlever les éléments marquants du protocole, mais aussi toutes les caractéristiques statistiques identifiables de la communication. Les principaux projets d'obfuscation par randomisation sont utilisés dans les plugins de transport de Tor [6]. Par exemple, obf-

6. `https://www.torproject.org/docs/pluggable-transports`

sproxy [Tor team 2017] a dans un premier temps (obfs2 [Winter 2017a]) enlevé tous les éléments marquants statiques de sa communication. Il ciblait les parefeux utilisant des règles de reconnaissance par expressions régulières statiques *i.e.* de blacklist. Dans sa version 3, obfs3 [Winter 2017b] a amélioré sa randomisation pour contourner les parefeux faisant une inspection en profondeur des paquets (DPI). Cependant, les parefeux qui font une analyse d'entropie de la communication ou d'autres mesures statistiques, permettent d'identifier une communication obfusquée par ce biais. Ces contremesures ont été présentées dans [Hjelmvik 2010]. Elles se basent sur des éléments marquants des échanges cryptographiques (tels que la négociation des clés de chiffrement) pour identifier les différentes communications. Pour pallier à ce problème, Winter *et al.* ont proposé ScrambleSuit [Winter 2013]. Cet outil implémente les précédentes obfuscations introduites dans obfsproxy mais ajoute en plus la randomisation dans la séquence de messages visant l'établissement de la communication chiffrée, *i.e.* l'échange des clés. Enfin, d'autres mécanismes de protections (ne relevant pas de l'obfuscation) contre l'*active probing* ont été implémentées dans obfs4 [Tor team 2017]. En effet, le client doit prouver au serveur qu'il connait une clé publique du serveur avant de négocier la clé de chiffrement des communications. Le coût de ces protections en terme d'augmentation de trafic reste assez faible.

2.3.2 Imitation

Le but de l'obfuscation par imitation est d'encoder les données de façon à ce qu'elles ressemblent à un protocole légitime. Par exemple, StegoTorus [Weinberg 2012] encode des données dans un jeu prédéfini de messages HTTP en utilisant des techniques de stéganographie. Comme *obfsproxy* ou *ScrambleSuit*, il est implémenté en tant que plugin de transport de Tor. De plus, une des particularités de ce projet est le découpage des flux Tor sur plusieurs connexions TCP, afin d'une part de réduire l'identification de la connexion par mesure de la taille des paquets, et d'autre part de rendre la reconstruction du flux réseau plus complexe. Le second projet le plus utilisé est SkypeMorph [Mohajeri Moghaddam 2012] qui s'appuie sur le fait que le protocole Skype est chiffré, ce qui lui permet à son tour de chiffrer ses communications. Il focalise alors son imitation sur les caractéristiques résiduelles (hors données chiffrées) d'une communication Skype légitime. Cependant, dans [Geddes 2013], les auteurs démontrent que ces communications peuvent quand même être identifiées à partir des liens sémantiques d'une communication, *i.e.* la grammaire du protocole et des champs ayant une dépendance inter-message, qui ne sont pas correctement gardés.

Comparé à la *randomisation*, le coût de l'*imitation* est plus élevé en termes de consommation mémoire ou en temps d'execution, mais aussi en termes de difficultés de déploiement. Cependant, cette technique permet de contourner les parefeux basés sur des whitelists de protocoles autorisés.

2.3.3 Tunneling

Le but de l'obfuscation par tunneling est d'utiliser un protocole et une communication légitime comme nouveau canal de communication, et d'encapsuler le flux de données à obfusquer dans ce canal. Cette technique peut être intégrée en utilisant une bibliothèque implémentant ce protocole légitime. Ainsi, les caractéristiques sont réellement celles d'une communication légitime. Cependant, le coût d'une telle obfuscation est encore plus élevé que pour l'imitation car les contraintes à respecter sont d'autant plus importantes. Skype et d'autres protocoles de VoIP ont encore une fois été largement utilisés pour le tunneling avec des projets tels que Freewave [Houmansadr 2013] ou Facet [Li 2014]. Ces projets ont été suivis par CensorSpoofer [Wang 2012] qui déploie une communication assymétrique. En effet, le flux descendant utilise une communication VoIP tandis que le flux montant utilise un canal de messagerie électronique. Cela lui permet de bénéficier d'un flux descendant beaucoup plus rapide tandis que le flux montant, généralement moins utilisé, garde un bas débit. Un des défauts majeurs du tunneling est le surcoût en consommation mémoire. Dans [Bridger 2015], Bridger *et al.* proposent l'utilisation des jeux vidéo en ligne pour contourner ces difficultés mais aussi pour assurer une cohérence plus importante des caractéristiques de la communication obfusquée avec la communication légitime. En effet, les jeux vidéos en ligne nécessitent des échanges réguliers et importants de données, notamment pour mettre à jour la position des joueurs et recevoir les actions du joueur. La part de données utiles de ces messages est maximisée dans ce genre d'application (pour limiter la charge du serveur). De plus, la grammaire du protocole est généralement simple pour les messages de position des joueurs et de leurs actions. Ainsi, il est plus facile de garder une cohérence des communications avec le protocole légitime que pour des protocoles complexes ou avec peu d'échanges réguliers. Cette cohérence est importante. En effet, le constat de Geddes *et al.* [Geddes 2013] est d'abord valable pour le tunneling. De plus, cette technique étant une évolution de l'*imitation*, elle aussi est efficace contre des parefeux basés sur des listes blanches "whitelists" de protocoles autorisés.

2.3.4 Programmable

Le but de l'obfuscation programmable est d'associer les avantages de la randomisation (faibles coût d'obfuscation) à ceux de l'imitation (se faire passer pour un autre protocole). En effet, le coût de déploiement d'une imitation vers un nouveau protocole est élevé en termes de nouveau développement et de maintenance. Ainsi, l'outil FTE a été proposé dans [Dyer 2013] pour automatiser ce processus. L'obfuscation est telle qu'elle respecte le format de message d'un protocole légitime décrit sous forme d'expression régulière. Cependant, d'une part, les caractéristiques de la communication ne sont pas reproduites, d'autre part la spécification du protocole imité restait trop simpliste. Ainsi, ils ont développé Marionette [Dyer 2015] pour répondre à ces limitations. L'outil *FTE* est ce qui se rapproche le plus de ce que l'on souhaite réaliser en proposant des obfuscations automatiques sur le format du

message. Cependant, il cherche à se rapprocher d'un autre protocole existant ne complexifiant pas sa compréhension globale. Ainsi, il pratique bien la *dissimulation* de protocoles et non pas la *confusion* de protocoles.

2.3.5 Cryptographie

Le chiffrement est un type de randomisation particulier qui fournit aussi des propriétés de sécurité : confidentialité et intégrité. Le trafic chiffré est très complexe à analyser par rétro-conception, sa puissance est alors très élevée. Cependant, le coût de déploiement, gestion, révocation des clés de chiffrement n'est pas négligeable. De plus, le coût en temps d'exécution est généralement important lors de l'utilisation d'algorithmes de chiffrement. [Wang 2015] propose des techniques pour identifier et catégoriser le trafic réseau même chiffré et qui peuvent être intégrées à des parefeux. De plus, l'hypothèse que l'attaquant n'a pas accès à l'application n'est pas toujours valide. Par exemple, les outils *Reformat* [Wang 2009] et *Dispatcher* [Caballero 2013] sont capables en analysant une application binaire, d'identifier automatiquement les interfaces des bibliothèques de chiffrement dans l'application, et à l'aide d'un débogueur, d'identifier le message en clair pour pratiquer une rétro-conception de protocole classique sur ce message en clair.

2.4 Conclusion

Dans ce chapitre, nous avons présenté différentes approches d'obfuscations de logiciels et de protocoles ainsi que leurs utilisations les plus classiques : rendre difficile la détection (*dissimulation*), complexifier la rétro-conception (*confusion*). Concernant le logiciel, les deux approches ont été étudiées de manière similaire et ont connu de nombreuses évolutions. En revanche, les obfuscations de protocoles de communication ont principalement pour objectifs de rendre la détection et l'identification de protocoles plus difficiles. Comme l'ont souligné Dyer et *al.* [Dyer 2015], l'obfuscation de protocoles a majoritairement ciblé le contournement de pare-feux et n'a pas été conçue pour être robuste face à de nouvelles techniques de rétro-conception de protocoles. Par exemple, nous avons présenté les outils *Dispatcher* [Caballero 2013] et *ReFormat* [Wang 2009] dans le chapitre 1.4.2 qui permettent l'identification automatique des interfaces de chiffrement pour effectuer la rétro-conception de protocoles chiffrés quand l'attaquant à accès à un client légitime. Or dans l'obfuscation logicielle (cf. section 2.2), la protection se fait non seulement en utilisant des techniques de *dissimulation* mais surtout en utilisant des techniques de *confusion* qui ont jusque-là été laissées de côté dans le domaine des protocoles. Ainsi, dans le chapitre 3, nous présentons une nouvelle méthode d'obfuscation de protocoles. Contrairement aux solutions majoritairement rencontrées, dans lesquelles on se protège contre la *détection*, notre approche va spécifiquement chercher à se protéger contre la rétro-conception, *i.e.* présenter des obfuscations de *confusion*. Cette méthode utilise notamment les challenges de la rétro-conception afin de réaliser des obfuscations au niveau de la spécification du protocole.

Chapitre 3

Obfuscations de spécification de protocole

Sommaire

Une obfuscation transforme un composant P en un composant P' assurant la même fonctionalité mais dont la rétro-conception est plus complexe. Inévitablement, l'obfuscation et la rétro-conception sont intimement liées. En effet, les challenges de la rétro-conception peuvent servir l'obfuscation. En particulier, dans ce chapitre, nous formalisons des obfuscations qui s'appuient sur les challenges de la rétro-conception suivants : le découpage du message en champs, la classification des messages et le choix du modèle pour l'inférence. Ces obfuscations concernent le format du message uniquement, l'obfuscation de la grammaire du protocole n'est pas abordé dans ce mémoire. Ce chapitre présente le modèle que nous avons adopté pour la représentation du format du message ainsi que des différentes obfuscations utilisées.

3.1 Modèle de format de messages

Dans le domaine de la spécification de protocoles, il n'existe pas de standard communément établi. En effet, les formats de message sont généralement adaptés et spécialisés aux besoins spécifiques des protocoles. Toutefois, ils sont souvent structurés sous la forme d'un arbre. Un *AST* (*Abstract Syntaxic Tree*) est donc adapté à

la représentation des messages. À notre connaissance, les formats existants n'ont pas été conçus pour subir des obfuscations. Nous avons décidé de proposer un nouveau modèle plus adapté pour l'obfuscation des formats de message et pour la génération automatique du code permettant de sérialiser (i.e., le *serialiseur*) et parser (i.e., le *parseur*) les messages.

La stratégie d'obfuscation que nous proposons doit tenir compte de ce fait pour pouvoir être employée pour différents protocoles. Notre modèle est basé sur des grammaires qui contrôlent la structure des messages stockés sous la forme d'AST. Il peut représenter une grande variété de formats de message. Plutôt que de décrire le format obfusqué d'un message, l'utilisateur de notre stratégie d'obfuscation doit simplement décrire ce format au travers de la première grammaire. Par transformation automatique, notre stratégie consiste à générer d'autres grammaires, de plus en plus complexes à rétro-concevoir.

Dans la suite, la grammaire sera nommée *Graphe de Format de Message* (GFM) et représente tous les messsages manipulés par le protocole. Un AST est une instance d'un message du GFM. Une feuille de l'AST représente la valeur d'un champ du message. Le message complet est construit en concaténant les feuilles de l'AST durant un parcours récursif descendant de l'AST.

3.1.1 Définitions

```
type t_ast =
| Ast_leaf of bytes
| Ast_sequence of t_ast list
| Ast_repetition of t_ast list
| Ast_star of t_ast list
| Ast_choice of t_ast
```

FIGURE 3.1 – Structure représentant un AST

Un AST est représenté par un type algébrique, présenté dans la figure 3.1. Les données sont stockées via le constructeur `Ast_leaf`. Nous retrouvons dans ce type toutes les constructions habituelles des langages réguliers : concaténation (via le constructeur `Ast_sequence`), répétition (via les constructeurs `Ast_repetition` et `Ast_star`) et le choix entre différentes alternatives (via le constructeur `Ast_choice`). `AST_repetition` contrairement à `Ast_star` possède un nombre défini d'éléments. Ces constructeurs sont nécessaires car les formats de message sont habituellement de nature régulière (cf. chapitres précédents).

Un GFM est également représenté par un type algébrique, présenté dans la figure 3.2. Nous retrouvons à nouveau les mêmes constructions que pour l'AST, avec des nuances que nous spécifierons plus tard. Plusieurs constructeurs nouveaux sont ajoutés : le constructeur `t_dissect` contenant `Gfm_len`, `Gfm_len_fixed`, `Gfm_delim` et `Gfm_delim_fixed` ; et le constructeur `Gfm_fct` servant de support

```
type t_path = int list
type t_dissect =
| Gfm_len of t_path
| Gfm_len_fixed of nat
| Gfm_delim of t_path
| Gfm_delim_fixed of bytes
type t_gfm =
| Gfm_leaf of t_dissect
| Gfm_sequence of t_gfm list
| Gfm_repetition of t_path * t_gfm
| Gfm_star of t_dissect * t_gfm
| Gfm_choice of t_path * nat * t_gfm * t_gfm
| Gfm_fct of fct
```

FIGURE 3.2 – Structure représentant un GFM

à certaines obfuscations et correspondant à l'exécution d'une fonction. Les premiers contrôlent la manière dont la dissection des champs est effectuée : soit via une taille pour `Gfm_len` (la taille est fournie via un lien vers un autre noeud) et `Gfm_len_fixed` (la taille est fournie directement dans le constructeur), soit via un délimiteur pour `Gfm_delim` (le délimiteur est fourni via un lien vers un autre noeud) et `Gfm_delim_fixed` (le délimiteur est fourni directement dans le constructeur).

Le paramètre `t_path` correspond à un lien vers un précédent champ. La dissection d'un champ a (en taille ou par un délimiteur) peut donc dépendre d'un champ b précédent (qui contient la taille de a si a est `Gfm_len` ou qui contient le délimiteur de a si a est `Gfm_delim`). Si la taille ou le délimiteur sont fixés dans la spécification du protocole, alors les éléments `Gfm_len_fixed` et `Gfm_delim_fixed` sont utilisés. La différence entre `Gfm_repetition` et `Gfm_star` est liée à la manière de délimiter les données : `Gfm_repetition` possède un lien vers un champ indiquant le nombre d'élements de cette répétition alors que `Gfm_star` possède un dissecteur permettant de retrouver la taille disponible pour parser un nouvel élément. Nous ajoutons une restriction supplémentaire sur le dissecteur de `Gfm_star` : il ne peut pas être `Gfm_len_fixed`. En effet, cela reviendrait à définir un champ dont le nombre d'éléments est dynamique, mais dont la taille est fixée par le protocole, *i.e.* statique. Même s'il serait possible de créer un tel protocole, son utilisation impliquerait de fortes contraintes pour éviter les erreurs.

3.1.2 Exemple

Dans cette section, nous considérons un exemple qui est dérivé du protocole *Tcp-Modbus* [Swales 1999], où la taille des données finales dépend d'un champ `Size` au lieu d'avoir une taille fixe dans la norme *Tcp-Modbus*. Nous considérons deux types de messages M_1 et M_2. Le premier champ de ces deux types, de 1 octet, correspond à un mot clé (Function Code, *i.e.* `Fc`) permettant d'identifier le type.

Le second champ, de 2 octets, contient des données (`Ref`). En fonction de la valeur du premier champ, la suite de la structure diverge. Pour les messages de type M_1, la suite contient un tableau (`Tab`) de couples `Size1`/`Data1`. La longueur de chaque champ `Data1` est fournie par le champ `Size1` qui précède. Pour les messages de type M_2, la suite contient simplement une taille (`Size2`) et une donnée (`Data2`). Le graphe GFM est donné en FIGURE 3.3. Par souci de lisibilité, les *peignes* du graphe sont aplatis, ainsi, les nœuds `Fc`, `Ref` et `O` sont tous les trois des enfants de `M`. Les paramètres `t_dissect` sont représentés dans la case en bas à droite du nœud. S'il n'y a pas de `t_dissect`, la case est omise par souci de lisibilité. Les paramètres `t_path` sont représentés à la fois par une flèche en pontillés et par le nom du nœud entre parenthèse, exemple "Rep(Count)" du nœud `Tab`. Le nom des types de nœud est raccourci comme suit par souci de lisibilité des figures :

- `Gfm_leaf` → `Leaf`
- `Gfm_sequence` → `Seq`
- `Gfm_repetition` → `Rep`
- `Gfm_star` → `Star`
- `Gfm_choice` → `Choice` avec sur les flèches vers les sous-nœuds la valeur associée à cette branche
- `Gfm_len` → `Len` avec entre parenthèses le nom du nœud associé
- `Gfm_len_fixed` → `LenF` avec entre parenthèses la taille de ce nœud
- `Gfm_delim` → `Delim` avec entre parenthèses le nom du nœud associé
- `Gfm_delim_fixed` → `DelimF` avec entre parenthèses le délimiteur de ce nœud

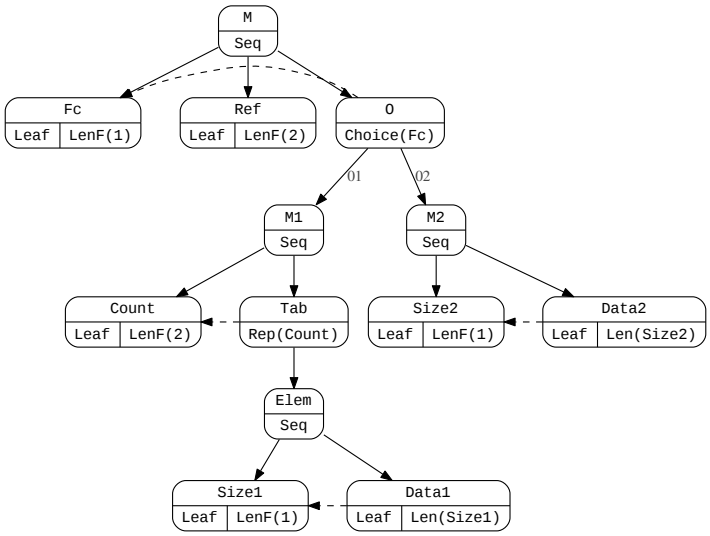

FIGURE 3.3 – Exemple de graphe de format de message comportant deux messages différenciés par le mot clé *Fc*.

Ce GFM peut être instancié par différents AST. Deux exemples pour les messages M_1 et M_2 sont donnés en figure 3.4 et figure 3.5. Un AST instancie une valeur pour le champ `Fc`, qui détermine ainsi le choix associé au nœud `O`. Il est donc normal qu'une des branches de ce choix n'apparaisse pas dans les différents AST. Le message M_1 contient deux éléments dans son tableau, chaque élément étant une séquence de `Size1`/`Data1`. La concaténation des champs terminaux donne la séquence d'octets suivante : `0100080002026a6a036b6b6b`. Notons que la valeur du champ `Count` est `0x02` mais, comme le nœud du GFM associé est `Gfm_len_fixed 2` (avec une taille fixée à deux octets), la sérialisation de ce champ aboutit à la valeur `0x0002`. De la même manière, le message M_2 contient une unique séquence de `Size2`/`Data2` et la concaténation des champs terminaux donne la séquence d'octets suivante : `020009046c6c6c6c`. Dans les AST, les valeurs associées au type `t_path` sont omises, mais nous avons choisi de faire figurer les dépendances par des pointillés. La fonction de sérialisation/parsing parcours l'AST d'après le GFM, il est donc possible de retrouver la valeur permettant de choisir la branche du `Gfm_choice`. Par exemple, dans le message de M_1, lors de la sérialisation (ou parsing) du champ `O`, les champs `Fc` et `Ref` ont déjà été sérialisés (ou parsés). Pour faire référence à `Fc`, la valeur de type `t_path` du GFM à utiliser est donc `[0x01]`. Le contenu des champs est représenté dans la case sous `Leaf`.

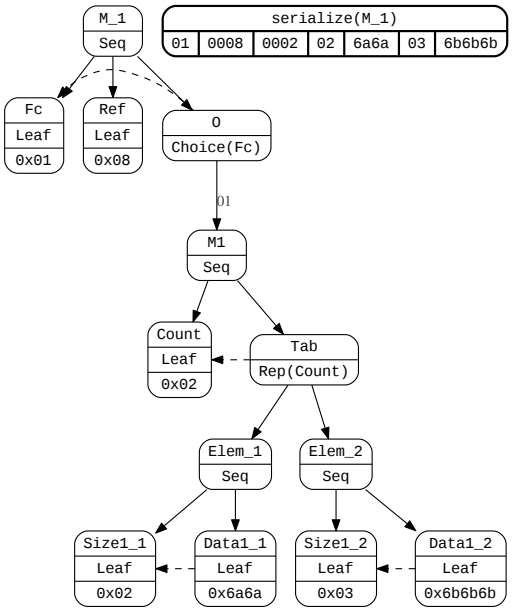

FIGURE 3.4 – AST d'un message M_1

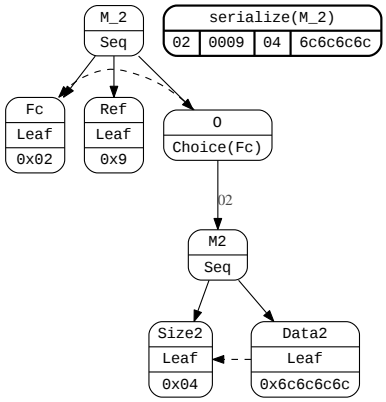

FIGURE 3.5 – AST d'un message M_2

3.1.3 Parseur et serialiseur

Un GFM contrôle la manière de sérialiser un AST en une séquence d'octets et parser une séquence d'octets vers un AST. Par la suite, nous utiliserons les racourcis de langage : parser un message ou sérialiser un message. Sérialiser un message correspond à l'analyse d'un AST pour en déduire la séquence d'octets associée ou rejeter cet AST s'il n'est pas conforme au GFM. Parser un message est l'opération inverse et traduit une séquence d'octets en AST ou rejette cette séquence si elle n'est pas conforme au GFM.

La fonction `serialize` possède trois paramètres : `mem`, `gfm` et `ast`. Le paramètre `ast` correspond à l'AST à sérialiser. Le second paramètre `gfm` correspond au GFM auquel l'AST doit se soumettre. Le premier paramètre, `mem` correspond aux parties de l'AST déjà sérialisées. Cette mémorisation, via `mem`, permet de gérer les liens entre les champs (par exemple, un champ contenant un entier indiquant une longueur et le champ qui doit posséder cette longueur). Le paramètre de type `t_path` des constructeurs de `t_gfm` correspond à un parcours de cette mémorisation. La fonction `serialize` navigue de manière synchronisée au sein de l'AST et du GFM pour réaliser des invocations récursives. Autrement dit, elle effectue un parcours en profondeur de l'AST pour en dériver une séquence d'octets correspondant au message.

La fonction `parse` possède également trois paramètres : `mem`, `gfm` et `msg`. Le paramètre `msg` correspond au message à parser. Le paramètre `gfm` correspond au GFM avec lequel le message doit être parsé. Le premier paramètre, `mem` joue le même rôle que son homonyme dans la fonction `serialize`. La fonction `parse` analyse un message en parcourant récursivement le GFM associé. Pour un nœud du GFM, la fonction découpe la séquence courante d'après le `t_dissect` de ce nœud, puis elle analyse chaque nœud du sous-graphe de ce nœud. Les morceaux issus du découpage et de l'analyse sont ensuite assemblés sous la forme d'un AST. Si l'analyse ou la

dissection d'un nœud dépend d'un autre nœud (via un `t_path`), alors ce lien doit être dirigé vers un nœud qui a déjà été analysé, *i.e.*, il ne peut pas y avoir de lien en avant dans le GFM.

Pour assurer le bon fonctionnement du programme, il faut s'assurer que les messages sérialisés à partir d'un AST peuvent être parsés vers le même AST et inversement. Dans la section 3.3.1, nous validons formellement ces fonctions avec le soutien du programme Coq

Dans le chapitre 4, nous proposons une implémentation où le parseur et le serialiseur ne sont pas génériques mais sont spécialisés pour un GFM donné. Nous avons fait ce choix pour que chaque code d'implémentation soit différent et nécessite un nouveau travail de rétro-conception de la part d'un attaquant.

3.2 Obfuscations des GFM

Une obfuscation (τ : `t_gfm` $\rightarrow$ `option t_gfm`) transforme un GFM en un autre GFM tout en garantissant l'inversibilité. Dans la suite, une obfuscation sera également nommée transformation. L'ensemble des transformations doit être applicable à un large éventail de GFM. Par conséquent, plutôt que d'obfusquer tout un GFM via une unique transformation, nous proposons plusieurs transformations atomiques, qui se focalisent chacune sur des motifs d'un GFM et qui peuvent être composées.

Une transformation opère donc à partir d'un nœud du GFM. Elle peut s'appliquer à ce seul nœud, ou à un sous-graphe de ce nœud. Certaines transformations vont plus loin et s'appliquent au nœud et au sous-arbre de ce nœud, *i.e.*, le nœud et tous ses nœuds enfants. Ce nœud est identifié par un chemin depuis la racine (à l'instar d'un parcours dans `mem` avec une variable du type `t_path`). Tous les nœuds de ce chemin correspondent aux parents du nœud qui va subir la transformation. Les nœuds parents et la structure du sous-graphe associé au nœud qui va subir la transformation doivent respecter des conditions particulières pour assurer l'inversibilité de la transformation. Ces conditions sont formulées sous la forme de contraintes générales présentées dans la section suivante.

Nous avons classé les différentes transformations en fonction de leur champ d'application. D'après la taxonomie des obfuscations [Collberg 1997], les obfuscations de données peuvent être de différents types :

— *Sauvegarde et encodage des données* - modifie le type de stockage ou l'encodage des données. Pour les protocoles, cela correspondrait à la couche de chiffrement ou à l'encapsulation d'un protocole à protéger dans une surcouche protocolaire.

— *Agrégation* - modifie la manière dont les données sont reconstruites. Pour les protocoles, cela correspond à modifier la reconstruction des champs. Par exemple, ils peuvent être fusionnés, découpés, obtenus par calculs algébriques...

— *Ordonancement* - modifie l'ordre dans lequel les données sont stockées. Pour les protocoles, cela correspond à une modification de l'ordre des champs du

protocole ou de leur méthode de dissection.

Ainsi, nous proposons des obfuscations de protocole qui ciblent l'**agrégation** ou l'**ordonnancement**. Chaque transformation peut ainsi appartenir à la catégorie des transformations d'agrégation ou à la catégorie des transformations d'ordonnancement. Une transformation d'agrégation change la manière dont un nœud est sérialisé. Une transformation d'ordonnancement change soit la manière dont les nœuds sont disséqués soit l'ordre dans lequel ils sont parsés.

Lors des illustrations des transformations, nous présentons leur impact sur un AST exemple. Nous faisons figurer les informations du GFM (dissection `t_dissect`, et obfuscation `Gfm_fct` nommée par son obfuscation associée) dans l'AST pour un souci de lisibilité et de compréhension.

3.2.1 Contraintes sur les transformations

Pour assurer qu'un AST valide peut correctement être sérialisé et qu'un message valide peut correctement être parsé, les transformations doivent respecter des contraintes qui sont générales à toutes les transformations que l'on propose. En particulier, une transformation appliquée à un nœud ne doit pas perturber le parsing des nœuds parents ni de lui-même. De plus, le parsing des enfants doit toujours être possible. Le point le plus délicat correspond aux délimiteurs. En effet, il y a de forts risques que la transformation crée artificiellement un délimiteur servant à disséquer un nœud. Supposons un nœud `Gfm_delim`, délimité par la séquence `abc` et contenant un nœud `Gfm_leaf`, nommé `n` par la suite. Un AST valide vis-à-vis de ce GFM peut contenir, par exemple, la valeur `12345678` pour le nœud `n`. La sérialisation de cet AST aboutit au message `12345678abc`. Si la transformation s'applique au nœud `n` et modifie par exemple la sous-séquence `567` en `abc`, alors le résultat de la sérialization devient `1234abc8abc`. Cette transformation ne modifie pas la taille du message résultant. Par contre, lors du parsing du message, le nœud sera dans un premier temps délimité avec la séquence `abc` pour ainsi isoler les données `1234`. Ces données ne correspondent pas aux données initiales, même après avoir tenté d'inverser la transformation. Cet exemple explique la raison du rejet d'un chemin contenant un nœud `Gfm_delim(_fixed)`. Nous pourrions, bien entendu, créer une autre famille pour laquelle les transformations garantissent de ne pas créer, de manière artificielle, les délimiteurs [1]. Mais nous avons choisi de durcir les contraintes en tenant compte de cette difficulté. Ceci conduit aux deux contraintes C.1 et C.2 énoncées ci-après.

Contrainte 1 *Aucun des nœuds du sous-graphe sur lequel s'applique la transformation n'est délimité par une séquence (dissecteur* `Gfm_delim` *ou* `Gfm_delim_fixed`*, abrégé en* `Gfm_delim(_fixed)`*).*

1. Par exemple, en imposant à l'utilisateur de fournir une expression régulière correspondant aux contenus valides pour le nœud n : la transformation `567`→`abc` serait alors identifiée comme invalide.

Contrainte 2 *Aucun des parents du nœud sur lequel s'applique la transformation n'est délimité par un dissecteur* `Gfm_delim(_fixed)`*.*

Pour assurer un parsing correct, les liens spécifiés via `t_path` doivent être conservés. Plus particulièrement, un GFM contenant un lien en avant ne permet pas de faire une analyse syntaxique correcte des messages. Ainsi, la transformation ne doit pas créer après application de tels liens. Ceci aboutit à la contrainte C.3.

Contrainte 3 *Les transformations ne doivent pas créer de liens (*`t_path`*) en avant.*

Enfin, nous avons choisi que les transformations ne dépendent que du GFM indépendamment des ASTs qui peuvent être produits par ce GFM. En effet, une transformation modifie un GFM et peut aussi avoir un impact sur les ASTs, par contre l'impact sur les ASTs doit dépendre uniquement de leur structure et non des valeurs qu'ils contiennent. Par exemple, si un nœud possède une taille dynamique (`Gfm_len`), alors la transformation ne peut pas changer la taille de ce nœud sans devoir mettre à jour le contenu du nœud pointé par le `t_path`. Cela se traduit par la contrainte C.4.

Contrainte 4 *Une transformation appliquée à un sous-graphe ne doit pas modifier le contenu des nœuds pointés par des liens (*`t_path`*) en arrière.*

De plus, les transformations qui modifient la taille du nœud sur lequel elles s'appliquent doivent respecter une contrainte supplémentaire. En effet, une transformation ne doit pas modifier le contenu d'un nœud pointé par un `t_path` (C.4). Ainsi, aucun parent ne doit avoir un dissecteur `Gfm_len(_fixed)` car il faudrait mettre à jour cette taille en fonction des transformations (C.4). Comme les contraintes associées au `Gfm_delim(_fixed)` (C.1 et C.2) s'appliquent toujours, cela revient à dire qu'aucun parent ne doit avoir de `Gfm_dissect`, *i.e.* qu'aucun parent ne doit être de type `Gfm_star`, soit la contrainte C.5.

Contrainte 5 *Aucun des parents du nœud sur lequel s'applique la transformation n'est de type* `Gfm_star`*.*

3.2.2 Les transformations d'agrégation

Les transformations d'agrégation permettent d'obfusquer la donnée associée à un nœud. Par exemple, nous pouvons considérer un champ L qui correspond à la taille d'un champ D. En appliquant une transformation d'agrégation sur L, la valeur du contenu de L permettant de connaître la taille de D n'est plus directement lisible par les techniques classiques de rétro-conception. Un exemple d'AST, suivi de l'AST obfusqué avec une transformation de ce type (de la catégorie *SplitXor*) est représenté en figure 3.6.

Le tableau 3.1 résume l'ensemble des transformations d'obfuscation d'agrégation présentées dans cette thèse. Pour chaque transformation, la première colonne

indique le type d'obfuscation. La seconde colonne explique succintement l'effet de la transformation. La troisième colonne indique le challenge de la rétro-conception associé à cette obfuscation. Enfin, la dernière colonne donne le coût de l'obfuscation sur la taille du message sérialisé. Dans la suite, nous présentons chacune de ces obfuscations.

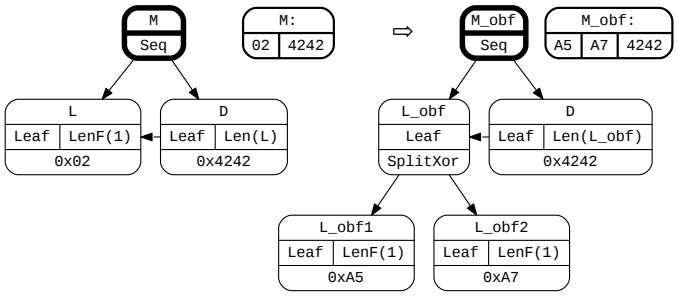

FIGURE 3.6 – Obfuscation *SplitXor* appliquée sur L. Les dissecteurs et fonction d'obfuscation associés au GFM sont affichés par souci de compréhension de l'AST.

3.2.2.1 ConstOp : ConstAdd, ConstSub et ConstXor

ConstOp est une transformation d'agrégation qui applique une opération inversible (addition, soustraction ou xor) sur le sous-arbre de n (jusqu'à ses feuilles), en s'appuyant sur une valeur générée aléatoirement au moment du déploiement (*const*). Si n contient des sous-nœuds, alors la transformation est appliquée de manière récursive à ces nœuds. Aussi, lors du parsing, pour retrouver ce sous-graphe, le contenu du nœud obfusqué doit tout d'abord subir l'opération inverse. La figure 3.7 présente un exemple d'application de cette transformation dans le cas d'une addition octet par octet avec la constante $0x02$. Cette transformation doit respecter les contraintes C.1 à C.4.

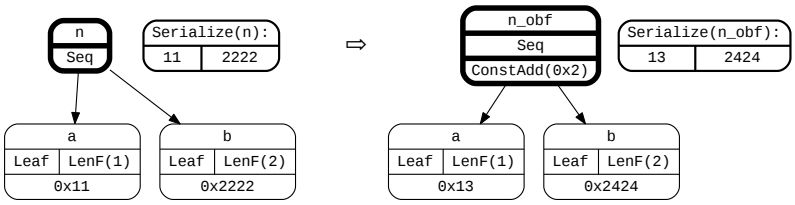

FIGURE 3.7 – Obfuscation *ConstAdd* appliquée sur n et une constante égale à 0x2.

Cette obfuscation s'applique à rendre le challenge de l'inférence du modèle plus complexe. En effet, les valeurs des champs ne sont plus directement lisibles dans le

Obfuscation	**Explications**	**Challenge**	**Coût**
ConstAdd, *ConstSub*, *ConstXor*	La sérialisation d'un nœud avec la valeur v est remplacée par la valeur $v_{obf} = Op(v, const)$ ($const$ est défini dans le framework).	Inférence du modèle et classification : les valeurs des champs ne sont plus directement lisibles dans le message sérialisé, un même AST peut aboutir à plusieurs messages différents.	0 octet
SplitAdd, *SplitSub*, *SplitXor*	La sérialisation d'un nœud avec la valeur v est remplacée par deux nœuds dont les valeurs sont v_obf1 et v_obf2 telles que $vOp(= v\_obf1, v\_obf2)$ (l'une des deux valeurs est choisie alléatoirement).	Inférence du modèle et classification : de nouvelles dépendances entre les champs sont créées, un même AST peut aboutir à plusieurs messages différents.	len(v)
SplitCat	La sérialisation d'un nœud avec la valeur v est découpée en une séquence de deux nœuds avec les valeurs v_obf1 et v_obf2 telles que $v = concatenate(v\_obf1, v\_obf2)$.	Inférence du modèle et classification : opération non prise en compte par les outils semi-automatique de PRE.	0 octet
DupNode	Duplique un sous graphe n en un choix entre n_1 et n_2 en ajoutant un nœud k permettant de déterminer la branche à utiliser.	Inférence du modèle et classification : augmente le nombre d'encodages possibles pour un même nœud.	len(k) (1 octet)
TreeCrypto	Chiffre un nœud d'un graphe avec la fonction crypto C.	Inférence du modèle et classification : Le contenu du nœud n'est plus accessible en clair.	C(buf) - len(buf)

TABLE 3.1 – Tableau des transformations d'agrégation

message sérialisé. De plus, entre chaque déploiement du couple sérialiseur/parseur, la valeur constante *const* est potentiellement modifiée, ce qui permet d'introduire du polymorphisme de protocole. Cette transformation s'avère être particulièrement utile pour les nœuds contenant des mots-clés. Cependant, le mot-clé modifié reste constant pour un déploiement de protocole.

Les champs contenant des longueurs et des compteurs sont moins protégés de la rétro-conception par cette transformation. En effet, pour l'inférence réseau, des techniques utilisées dans l'outil Netzob [Guihery 2012b] permettent d'identifier de tels champs : la taille des données envoyées est modifiée (via l'application légitime) et une corrélation entre cette modification de taille et une modification équivalente dans les données est recherchée. Pour l'inférence applicative, le flux des données est suivi par du *DTA* (*Data Tainting Analysis*), et les champs contenant des longueurs et des compteurs sont identifiés comme étant associés à des conditions de sortie de différentes boucles du programme. Ainsi, l'introduction d'un biais par cette obfuscation ne complexifie pas l'inférence des champs contenant des longueurs et des compteurs.

La taille du message sérialisé n'augmente pas car l'ensemble du champ subit la même opération sur chacun de ses octets. Cependant, il y a un coût en temps d'exécution, associé à l'opération à réaliser. Ce coût dépend de l'implémentation du parseur et du sérialiseur choisie. Il est donc difficilement évaluable de façon théorique. Nous proposons une évaluation pratique en chapitre 5 de l'implémentation proposée.

3.2.2.2 SplitOp : SplitAdd, SplitSub et SplitXor

SplitOp est une transformation d'agrégation qui transforme un sous-arbre n en une séquence de deux sous-arbres n_obf1 et n_obf2, liés par une opération mathématique (addition, soustraction ou xor). La sérialisation correspond à la concaténation de feuilles de ces deux sous-arbres. Pour obtenir le contenu d'un nœud initial de n, il faut appliquer l'opération inverse sur le contenu des deux nœuds associés de n_obf1 et n_obf2. A l'instar de la transformation *ConstOp*, la transformation inverse doit d'abord être appliquée avant de traiter le sous-arbre si le nœud n correspond à un sous-arbre. La figure 3.6 est un exemple de l'application de l'obfuscation *SplitXor* sur un nœud terminal L qui est la cible d'une dépendance (la taille du champ suivant). La figure 3.8 présente un exemple d'application de *SplitAdd* avec, pour le contenu des nœuds en $\_obf2$, une séquence d'octets s_i telle que $s_i = i \mod 256$. Pour cette transformation, la contrainte sur les dépendances (C.3) se traduit par le fait qu'aucun de ses sous-nœuds ne doit avoir de dépendances à l'intérieur de ce sous-graphe. En effet, cette transformation découpe un sous arbre en deux sous-arbres, ainsi, tous les nœuds du sous-arbre sont découpés et réparties entre les deux sous-arbres créés. Si un nœud avait une dépendance (`t_path`) à l'intérieur du sous-arbre, cette dépendance serait répartie sur les deux sous-arbres, et donc avec un lien en avant, ce qui est interdit par C.3. Cette transformation doit respecter les contraintes C.1 à C.4 et C.5.

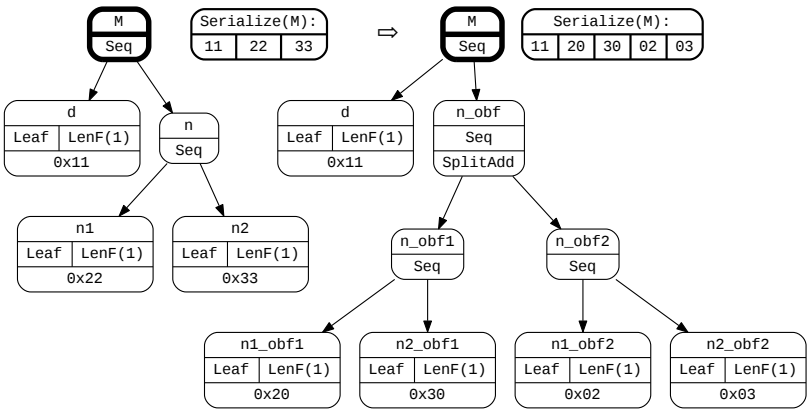

FIGURE 3.8 – Obfuscation *SplitOp* appliquée sur le nœud n.

Cette obfuscation s'applique à rendre le challenge de l'inférence du modèle plus complexe. En effet, des dépendances supplémentaires entre champs sont créées, ce qui augmente la difficulté d'avoir un modèle correct sur l'ensemble du message. Tout comme l'obfuscation *ConstOp*, cette obfuscation permet de modifier les mots-clés et d'introduire du polymorphisme de protocole. Cependant, comme la valeur des nœuds $\_obf1$ ou $\_obf2$ peut être choisie à l'exécution, un même AST peut engendrer plusieurs messages valides différents, à partir du même GFM. Ainsi, les techniques d'inférence de protocole (inférence réseau) basées sur la reconnaissance de motifs ne sont plus efficaces. De plus, les champs contenant une longueur ou un compteur ne peuvent plus être identifiés via une corrélation entre la variation de la taille des données et la variation de la valeur du champ contenant la longueur ou le compteur correspondant.

Cette obfuscation augmente significativement la taille du message sérialisé. En effet, un nouveau champ de la même taille que le résultat de la sérialisation du nœud n est créé pour pouvoir effectuer les opérations idoines. Elle a un coût en temps d'exécution et en ressources mémoire équivalent à l'obfuscation *ConstOp*.

3.2.2.3 SplitCat

SplitCat est une transformation d'agrégation qui transforme un nœud n en une séquence de nœuds n_obf1 et n_obf2 telle que la concaténation de ces deux nœuds aboutit au contenu du nœud n. Elle ne s'applique qu'à des `Gfm_leaf` de taille fixe `Gfm_len_fixed`. En effet, si le nœud avait une taille dynamique rien ne garantit que la taille soit suffisante (> 1) pour pouvoir être découpée en deux. La figure 3.9 présente un exemple d'application de cette transformation qui doit respecter les contraintes C.1 à C.4.

Cette transformation ne modifie clairement pas le message après sérialisation. Elle n'apporte donc rien pour l'obfuscation, lorsqu'elle est employée seule. Par

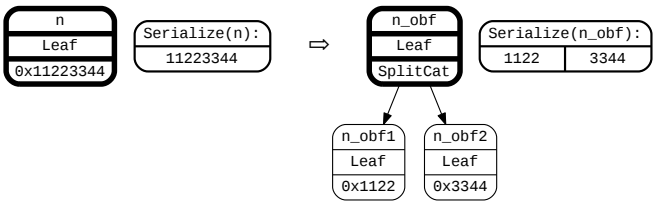

FIGURE 3.9 – Obfuscation *SplitCat* appliquée sur un nœud n.

contre, en combinaison avec les autres obfuscations, elle permet d'intercaler d'autres nœuds entre n_obf1 et n_obf2. Nous menerons donc une analyse en considérant que cette insertion a eu lieu.

Cette transformation vise à rendre le challenge de l'inférence du modèle plus complexe. En effet, non seulement des dépendances supplémentaires entre champs sont créées, ce qui augmente la difficulté d'avoir un modèle correct sur l'ensemble du message. Mais de plus, les techniques d'inférence de protocole supposent que les données des différents champs sont sérialisés de manière contigüe. Avec cette obfuscation, le champ est séparé en plusieurs suites d'octets qui peuvent être déplacées au sein du message sérialisé, notamment avec des transformations d'ordonnancement.

La taille du message sérialisé n'augmente pas avec cette obfuscation. Le coût de cette obfuscation en temps d'exécution et en ressources mémoire est a priori moindre que pour les obfuscations *SplitOp* ou *ConstOp* car il n'y a besoin que d'une recopie des données et pas de calcul.

3.2.2.4 DupNode

DupNode est une transformation d'agrégation qui transforme un nœud n en un nœud de type `Gfm_choice` o contenant deux nœuds n_obf1 et n_obf2 correspondant à des copies de n (les sous-arbres à chaque fois). Le choix entre ces deux nœuds se fait via la valeur d'un nœud k. La figure 3.10 présente un exemple d'application de cette transformation qui doit respecter les contraintes C.1 à C.5.

Cette obfuscation s'applique à rendre le challenge de l'inférence du modèle et la classification des messages plus complexes. En effet, pour un même nœud original n, il y aura deux nœuds optionnels créés n_obf1 et n_obf2. Dans un premier temps, ces nœuds sont identiques, mais au fur et à mesure de la composition des obfuscations, leur sous-graphe pourra varier. Cela aura pour conséquence de fournir plusieurs encodages possibles pour un même nœud. Ainsi, le nombre de clusters créés par la classification des messages augmentera en conséquence lors de la rétro-conception. Ainsi, chaque cluster aura un nombre plus restreint de messages et l'inférence des modèles s'effectuera sur un ensemble moins complet. Cela se traduit généralement par des erreurs de sur-apprentissage plus importante.

Le coût de cette obfuscation sur la taille du message sérialisé correspond à la

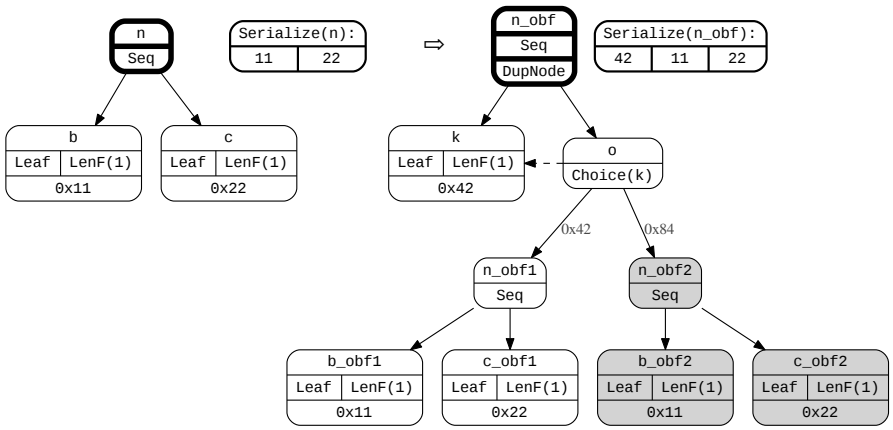

FIGURE 3.10 – Obfuscation *DupNode* appliquée sur n. La branche grisée correspond au choix qui n'est pas pris.

taille du champ k ajouté. De plus, le coût en temps d'exécution et en ressources mémoire varie peu car la suite du parsing se fait à partir d'une simple comparaison entre la valeur du nœud k et les valeurs de référence.

3.2.2.5 TreeCrypto

TreeCrypto est une transformation d'agrégation qui chiffre le résultat de la sérialisation d'un nœud. Cette transformation peut s'appliquer à l'intégralité du GFM (i.e., au nœud racine) ou à un sous-arbre de ce GFM. Du point de vue du domaine de l'obfuscation, cette transformation unifie les approches classiques de l'obfuscation avec la cryptographie. L'impact sur l'AST s'applique après la sérialisation du sous-graphe et non de manière récursive sur chacun de ses sous-nœuds. La figure 3.11 présente un exemple d'application de cette transformation sur un sous-graphe n. Si cette transformation change la taille du nœud, alors tout comme la transformation *DupNode*, elle doit respecter les contraintes C.1 à C.5. Sinon (transformation ne changeant pas la taille du nœud), elle n'a pas besoin de la contrainte C.5. Par exemple, l'algorithme RC4 [2] produit un chiffré de la même taille que le clair.

Cette obfuscation s'applique à rendre le challenge de l'inférence du modèle, la classification et la dissection des messages plus complexes, en bénéficiant des avantages de la cryptographie vis-à-vis de la confidentialité.

Le coût de cette obfuscation sur la taille du message sérialisé correspond à la surcharge liée à l'algorithme cryptographique employé. De plus, le coût en temps d'exécution et en ressources mémoire varie plus ou moins en fonction de l'algorithme employé et de la partie de l'AST sur laquelle il s'applique.

2. `https://sites.math.washington.edu/~nichifor/310_2008_Spring/Pres_RC4%20Encryption.pdf`

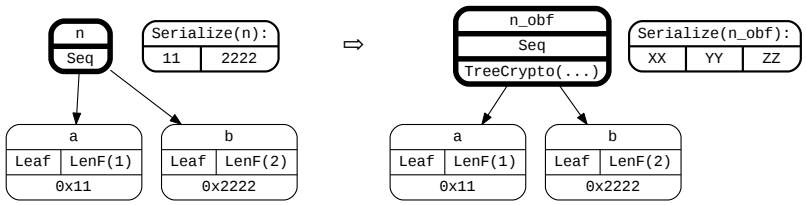

FIGURE 3.11 – Obfuscation *TreeCrypto* appliquée sur n. Le contenu des nœuds de l'AST n'est pas modifié dans l'AST mais au moment de la sérialisation il y a le chiffrement, d'où le résultat de la sérialisation en XXYYZZ différent du contenu de l'AST.

3.2.3 Les transformations d'ordonnancement

Les transformations d'ordonnancement permettent de modifier l'ordre avec lequel les différents champs sont sérialisés ou la façon dont la dissection des champs est effectuée.

Le tableau 3.2 résume l'ensemble des transformations d'ordonnancement présentées dans cette thèse. Pour chaque transformation, la première colonne indique le type d'obfuscation. La seconde colonne explique succintement l'effet de la transformation. Enfin, la dernière colonne indique le challenge de la rétro-conception associé à cette obfuscation.

3.2.3.1 BoundaryChange

BoundaryChange est une transformation d'obfuscation d'ordonnancement qui change la manière dont un nœud n est disséqué. Elle permet notamment de s'affranchir des délimiteurs (`Gfm_delim(fixed)`) qui constituent la majorité des contraintes des obfuscations présentées. Pour ce faire, il suffit de changer le nœud n par une séquence contenant la taille suivie des données. La figure 3.12 présente un exemple d'application de cette méthode. Cette transformation doit respecter les contraintes C.1 à C.5.

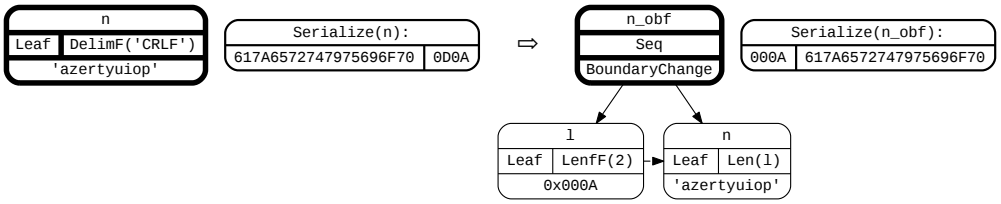

FIGURE 3.12 – Obfuscation *BoundaryChange* appliquée sur un nœud n. n est de type `Gfm_Leaf` mais il aurait pu tout aussi bien être de type `Gfm_star`.

Obfuscation	**Explications**	**Challenge**	**Coût**
BoundaryChange	Une délimitation par une séquence seq est rétirée ou remplacée par un champ longeur l si nécessaire.	Détermination de la délimitation des champs : les séquences classiques de fin de champs sont retirées.	len(l) - len(seq)
PadInsert	Un nœud p avec une valeur aléatoire est ajouté à une séquence.	Complexification de l'identification de la donnée utile.	len(p)
DissectFromEnd	La dissection du nœud se fait à partir de la fin du message.	Inférence du modèle et classification : une partie de la dissection du message se fait à partir de la fin du message et non plus classiquement de gauche à droite.	0 octet
ReadFromEnd	La dissection du nœud se fait à partir de la fin du message et la lecture du nœud se fait de gauche à droite.	Inférence du modèle et classification : une partie de la dissection du message se fait à partir de la fin du message et non plus classiquement de gauche à droite.	0 octet
RepSplit	Un nœud contenant une répétition d'une séquence est remplacé par une séquence de répétitions.	Inférence du modèle : change un modèle régulier $(ab)^*$ en une grammaire grammaire contextuelle $a^n b^n$.	0 octet
StarSplit	Un nœud contenant une répétition d'une séquence est remplacé par une séquence de répétitions.	Inférence du modèle : change un modèle régulier $(ab)^*$ en une grammaire grammaire contextuelle $a^n b^n$.	0 octet
ChildMove	Permutation de deux sous-nœuds d'une séquence.	Classification : les champs qui ont une importance pour le parsing ne sont plus forcément en début de message.	0 octet

TABLE 3.2 – Tableau des transformations d'ordonnancement

Le but premier de cette obfuscation n'est pas de rendre l'inférence du modèle plus complexe. Par contre, en enlevant les champs disséqués par des délimiteurs, le panel d'obfuscations applicables sur ces champs est plus large.

Le coût de cette obfuscation en terme de taille du message sérialisé correspond à la taille du délimiteur moins la taille du nœud ajouté. Dans l'exemple de la figure 3.12, nous avons arbitrairement fixé cette taille à 2, permettant de définir des champs de taille de 65ko. Une plus grande valeur est possible mais pour un protocole, cette taille est généralement suffisante. De plus, la dissection par délimiteur est généralement moins efficace d'un point de vue algorithmique que l'utilisation d'une taille. Ainsi, cette obfuscation a globalement un coût de déploiement faible voire *positif* dans certains cas spécifiques (exemple de la figure 3.12 où les tailles sont équivalentes).

3.2.3.2 PadInsert

PadInsert est une transformation d'ordonnancement qui ajoute un champ quelconque dans une séquence. Elle s'applique donc au sous-graphe constitué du nœud parent, et de ses nœuds enfants uniquement (nous ne considérons pas la descendance des enfants suivants). Il est à noter que le nœud ajouté doit pouvoir être disséqué indépendamment du reste du message, ce qui nécessite une dissection de taille fixe ou un nœud supplémentaire indiquant la taille de ce nœud. Les figures 3.13 et 3.14 présentent un exemple d'application de cette transformation avec respectivement, une insertion de padding à taille fixe, et une insertion de padding à taille dynamique. Le premier padding n_obf1 a une taille fixe, il est inséré entre deux nœuds qui ont une dépendance ($a1$ et $a2$). Le second padding possède une taille dynamique, il est inséré en fin de séquence. Les contenu des nœuds de padding est, pour cet exemple, pris arbitrairement. Cette transformation doit elle aussi respecter les contraintes C.1 à C.5.

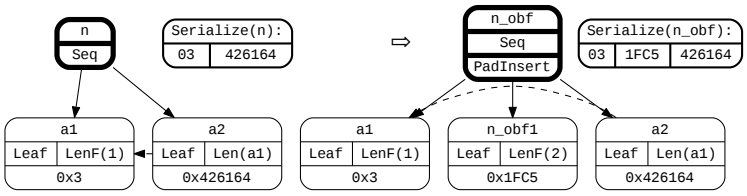

FIGURE 3.13 – Obfuscation *PadInsert* appliquée sur n, un nœud de padding (n_obf1) à taille fixe a été ajouté.

Cette obfuscation s'applique à rendre le challenge de l'inférence du modèle plus complexe. En effet, cela augmente le nombre de champs du message et permet potentiellement d'éloigner des champs qui auraient normalement des dépendances. Par exemple, un champ de données ne suivrait plus directement son champ taille. En utilisant des valeurs aléatoires, la classification des messages (pour l'inférence réseau)

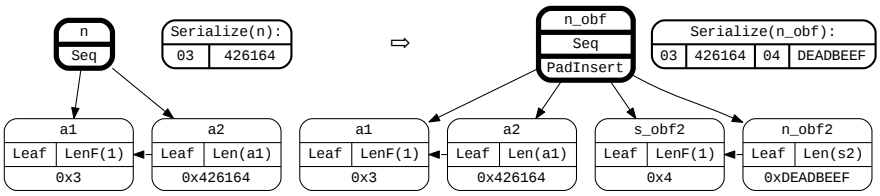

FIGURE 3.14 – Obfuscation *PadInsert* appliquée sur n, un nœud de padding n_obf2 à taille dynamique (s_obf2) a été ajouté.

peut être impactée négativement (deux AST identiques aboutissent à des messages sérialisés différents). A l'inverse, pour tenter de rendre l'inférence applicative plus complexe, ces nœuds peuvent être remplis avec des données qui proviennent de précédents messages. Il y aura deux conséquences importantes : 1) étant donné que l'inférence applicative est principalement basée sur l'analyse de teinte de données, des dépendances artificielles entre ces messages vont logiquement être créées et 2) si la partie du précédent message qui est copié ne correspond pas à une dissection normale de ce message, l'analyse pour l'identification des bornes des champs en sera elle aussi complexifiée.

Le coût d'une telle obfuscation sur la taille du message sérialisé est la longueur des champs qui auront été ajoutés. En principe, ces champs peuvent être ignorés après leur dissection.

3.2.3.3 DissectFromEnd

DissectFromEnd est une transformation d'ordonnancement qui change la manière dont un sous-arbre n est disséqué. L'application aux enfants se fait de manière récursive. Le décompte de dissection se fait à partir de la fin du message. La figure 3.15 présente un exemple d'application de cette transformation qui doit respecter les contraintes C.1 à C.4.

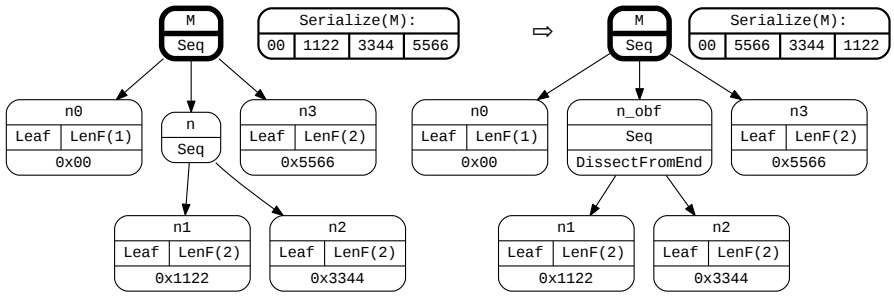

FIGURE 3.15 – Obfuscation *DissectFromEnd* appliqué sur le champ n

Cette obfuscation s'applique à rendre le challenge de l'inférence du modèle, la classification et la dissection des messages plus complexes. En effet, la lecture d'un message se fait classiquement de gauche à droite. Avec cette obfuscation, le parcours des champs du GFM du protocole se fait bien de gauche à droite ce qui permet de garder l'ordre dans lequel les champs sont parsés, mais dans le message sérialisé (*i.e.* le buffer), les champs sont répartis au début ou à la fin du buffer suivant la présence de l'obfuscation *DissectFromEnd*. Il n'y a pas à notre connaissance d'outil d'inférence réseau permettant d'analyser de tels messages. De plus, la classification des messages, notamment par l'outil *Veritas* [Wang 2011], s'appuie sur le fait que les champs significatifs sont souvent en début de message. Avec cette obfuscation, cette hypothèse devient caduque. Les outils d'inférence applicative ne sont à priori pas impactés par cette obfuscation car le *DTA* utilisé par ces outils permet de suivre l'utilisation des données quelque soit leur position dans le buffer. De manière plus générale, l'identification des bornes des champs (*i.e.* la dissection du message en champs) se trouve complexifiée pour les outils ne pratiquant pas d'inférence applicative.

Cette obfuscation ne change pas la taille du message sérialisé. De plus, le coût en temps d'exécution ou en ressources mémoire est négligeable car une dissection depuis le début ou la fin du message représente les même contraintes techniques dans la majorité des implémentations.

3.2.3.4 ReadFromEnd

Tout comme *DissectFromEnd*, pour *ReadFromEnd* le décompte de dissection se fait à partir de la fin du message. De plus, cette obfuscation inverse le sens de lecture des nœuds : une lecture depuis la fin du message. Cette transformation doit respecter les contraintes C.1 à C.4. Cette obfuscation possède les mêmes applications que *DissectFromEnd*, mais l'inversion du sens de lecture complique encore plus la tâche des outils d'inférence de protocole. Ainsi, le message sérialisé pour *DissectFromEnd* est `00|5566|3344|1122` tandis que pour *ReadFromEnd*, il devient `00|5566|4433|2211`.

3.2.3.5 RepSplit

RepSplit est une transformation d'ordonnancement qui découpe un `Gfm_repetition` T dont le sous-nœud est une `Gfm_sequence` e (contenant a et b pour l'exemple de la figure 3.16) en une `Gfm_sequence` e de `Gfm_repetition` T_obfa et T_obfb, dont le sous-nœud respectif est a et b. Cette transformation s'applique donc au sous-graphe constitué de T, e, a et b. Si a ou b possèdent des enfants, ils ne sont pas directement impactés. La figure 3.16 présente un exemple d'application de cette méthode. Dans le cas de cet exemple, la difficulté est de garder la dépendance de b vers le "bon" a. Dans l'implémentation, nous avons résolu ce problème en ajoutant un contexte de parcours de l'AST associé. Ainsi, $b2$ est bien associé à $a2$ grace à ce contexte. Cette transformation doit respecter les contraintes C.1 à C.4.

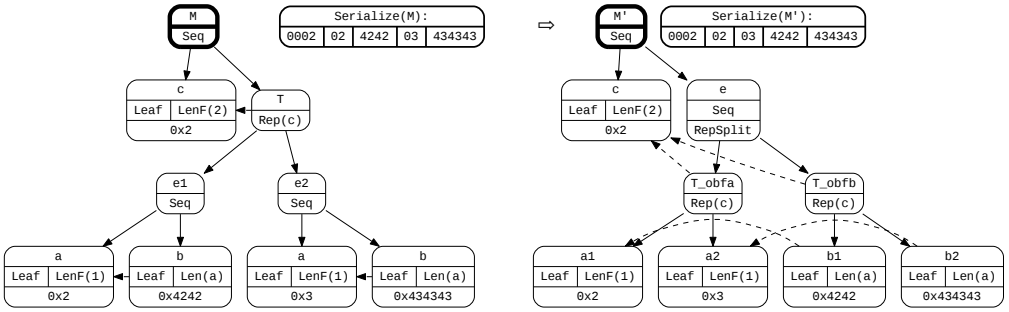

FIGURE 3.16 – Obfuscation *RepSplit* appliquée sur un nœud T. Le tableau contient 2 éléments. Chaque élément du tableau est indexé par son numéro.

Cette obfuscation s'applique à rendre le challenge de l'inférence du modèle plus complexe. En effet, elle modifie le type de modèle du message. Le message passe d'une grammaire de forme $(ab)^*$ (pour notre exemple de la figure 3.16) à une grammaire algébrique de forme $a^k b^k$ où k est une valeur qui est déterminée lors du parsing (contenu dans le champ c). Il n'y a pas à notre connaissance d'outil de rétro-conception de protocoles permettant l'inférence de tels modèles. Ainsi, le modèle qui serait appris par de tels outils ne serait qu'une approximation du modèle réel.

La taille du message sérialisé est inchangée. Cependant, le coût en temps d'exécution et en ressources mémoire sera plus important car il est nécessaire de parcourir une répétition pour la remplacer par deux répétitions avant la sérialisation (inversement lors du parsing).

3.2.3.6 StarSplit

La transformation *StarSplit* est très similaire à *RepSplit*. Elle découpe `Gfm_star` S dont le sous nœud est une `Gfm_sequence` e (contenant a et b pour l'exemple de la figure 3.16) en une `Gfm_sequence` e de `Gfm_star` S_obfa et S_obfb, dont le sous-nœud respecitif est a et b. De plus, chaque nouvelle `Gfm_star` possède une taille spécifique la et lb, ce qui change la taille globale du message sérialisé. Cette transformation s'applique donc au sous-graphe constitué de S, e, a et b. Si a ou b possèdent des enfants, ils ne sont pas directement impactés. La figure 3.17 présente un exemple d'application de cette méthode. Cette transformation doit respecter les contraintes C.1 à C.4.

Cette obfuscation possède les mêmes caractérisques que *RepSplit*. Cependant, la taille du message augmente car un nouveau nœud est ajouté afin de déterminer la taille du second `Gfm_star`.

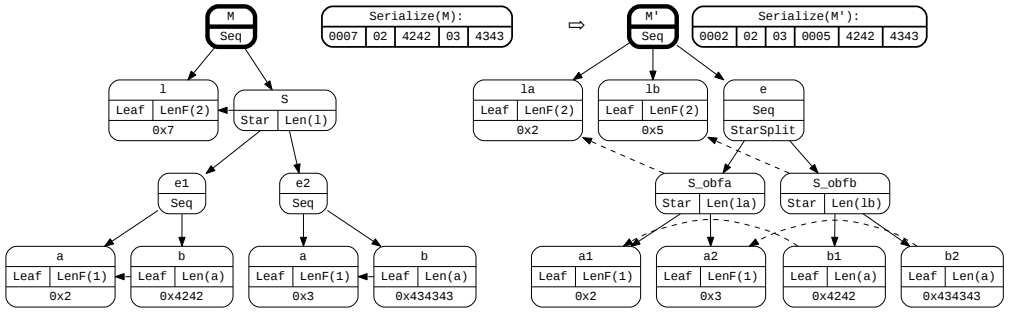

FIGURE 3.17 – Obfuscation *StarSplit* appliquée sur un nœud S. Le `Gfm_star` contient 2 éléments. Chaque élément est indexé par son numéro.

3.2.3.7 ChildMove

ChildMove est une transformation d'ordonnancement qui permute la place de deux nœuds $n1$ et $n2$ dans une séquence n du GFM. Cette transformation s'applique donc au sous-graphe n, $n1$ et $n2$. Une condition importante pour l'application de cette transformation est l'indépendance de $n2$ vis-à-vis du contenu de $n1$. De plus, durant la réalisation de cette transformation, il est nécessaire de mettre à jour les différents chemins de $n1$ et $n2$. Lors du parcours du GFM, un nœud ne peut pas avoir de dépendance vers un nœud qui n'a pas encore été parcouru (C.3). Cela se traduit pour cette transformation par le fait qu'aucun sous-nœud de $n2$ ne doit dépendre de $n1$. La figure 3.18 présente un exemple d'application de cette méthode. Cette transformation doit respecter les contraintes C.1 à C.4.

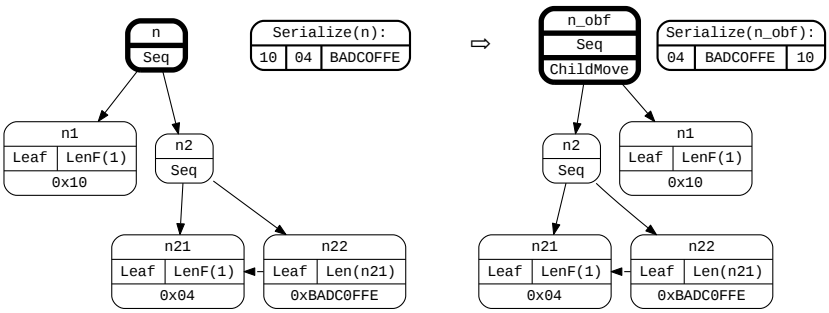

FIGURE 3.18 – Obfuscation d'un GFM avec un *ChildMove*

Cette obfuscation s'applique à rendre la classification des messages plus complexe. En effet, les champs déterminants du message pourront ne plus se situer en début de buffer sérialisé. Cependant, les champs déterminants pour la classification sont souvent les mots-clés. Or la suite du parsing est généralement liée à ces mots-clés qui ne doivent donc pas changer de place. Cette transformation est plus utile

si elle est combinée avec les autres.

Comme pour l'obfuscation *ReadFromEnd*, cette obfuscation ne change pas la taille du message sérialisé et son coût en temps d'exécution ou en ressources mémoire est négligeable car seul l'ordre change.

Synthèse

Le tableau 3.3 donne les contraintes auquelles les transformations doivent se conformer.

Transformation	C.1	C.2	C.3	C.4	C.5
ConstOp	X	X	X	X	
SplitOp	X	X	X	X	X
SplitCat	X	X	X	X	
DupNode	X	X	X	X	X
TreeCrypto	X	X	X	X	*
BoundaryChange	X	X	X	X	*
PadInsert	X	X	X	X	X
DissectFromEnd	X	X	X	X	
ReadFromEnd	X	X	X	X	
RepSplit	X	X	X	X	
StarSplit	X	X	X	X	X
ChildMove	X	X	X	X	

TABLE 3.3 – Récapitulatif des contraintes et des transformations. L'étoile indique que la transformation doit respecter la contrainte si elle change la taille du nœud.

3.2.4 Composition et application des obfuscations

Chaque transformation unitaire prise indépendamment aura une faible incidence sur l'obfuscation du GFM. Il faut composer ces obfuscations pour obtenir un meilleur résultat. Les exemples 3 à 5 en sont des illustrations :

Exemple 3 *L'obfuscation SplitCat n'a aucune incidence sur la sérialisation du message utilisée seule. Mais si elle est couplée avec ChildMove, alors les deux morceaux du champ seront séparés lors de la sérialisation.*

Exemple 4 *L'application à deux reprises de RepSplit permet d'augmenter sensiblement la complexité. Prenons pour exemple, un GFM contenant un nœud* `Gfm_repetition` *pour représenter le langage régulier* $(KSV)*$*. Si un AST contient 3 répétitions, alors le résultat de la sérialisation devient* $K_1S_1V_1K_2S_2V_2K_3S_3V_3$ *(les indices correspondent au numéro de l'élément dans le tableau). Les outils d'inférence devraient obtenir un modèle régulier qui s'approche de l'expression régulière de départ. Appliquons à présent une transformation RepSplit : nous obtenons le message* $K_1K_2K_3S_1V_1S_2V_2S_3V_3$ *qui n'est plus régulier. En appliquant à nouveau, nous obtenons le message* $K_1K_2K_3S_1S_2S_3V_1V_2V_3$ *qui est encore moins régulier. Ce genre de*

séquence est classiquement parsé par un automate à pile pour garder le lien entre les K_i, les S_i et les V_i. Cependant, le modèle qui sera obtenu par les outils d'inférence réseau sera de type $K^*S^*V^*$, sans le lien entre les K_i, les S_i et les V_i. (Certains outils d'inférence applicative peuvent trouver un modèle de type $K^nS^nV^n$, cependant, la multiple dépendance de K, S et V au même compteur n'est pas prévu pour la majorité d'entre eux).

Exemple 5 *L'application successive de RepSplit sur un GFM contenant un nœud* `Gfm_repetition` *pour représenter le langage régulier $(KV)*$ puis de DissectFromEnd sur le nœud K permet de créer une grammaire qui doit logiquement être parsée par un automate à pile. De la séquence d'origine $K_1V_1K_2V_2K_3V_3$, la sérialisation après RepSplit donne la séquence $K_1K_2K_3V_1V_2V_3$ puis après l'application de DissectFromEnd sur K : $V_1V_2V_3K_3K_2K_1$. Dans ce cas, les keywords apparaissent à la fin du message sérialisé.*

3.3 Validation Coq

Avec le support de l'outil d'aide à la preuve Coq, nous cherchons à vérifier : d'une part, qu'un message sérialisé à partir d'un AST peut être correctement parsé et inversement ; mais surtout, l'inversibilité des obfuscations présentées en section 3.2.

Afin de simplifier la tâche de validation du modèle, nous avons légèrement modifié le modèle présenté précédemment. Ce nouveau modèle est utilisé pour valider avec l'aide de l'outil Coq, l'exemple de code pour les fonctions `parser` et `serializer` génériques présentées en annexe A dans les figures A.1 et A.2, ainsi que les obfuscations. Nous nommons ce modèle, modèle Coq.

```
type t_ast =
| Ast_leaf of bytes
| Ast_sequence of t_ast * t_ast
| Ast_repetition of t_ast list
| Ast_star of t_ast list
| Ast_choice of t_ast
```

FIGURE 3.19 – Structure représentant un AST pour Coq.

Les séquences (`Ast_sequence`) ne contiennent que deux éléments, sinon, ils ne sont pas modifiés. Logiquement, cette simplification se retrouve sur les `Gfm_sequence` qui elles aussi ne contiennent que deux éléments. Pour le GFM, l'imbrication des types est enlevée pour simplifier la validation Coq. Ainsi, `Gfm_len(_fixed)` et `Gfm_delim(_fixed)` sont maintenant des types de `t_gfm`. De plus, comme les leafs doivent pouvoir être disséquées, le type est spécialisé en `Gfm_leaf` et `Gfm_leaf_fixed` : soit la taille est fixe et connue (ce qui est le cas la plupart du temps) et dans ce cas le type à appliquer est `Gfm_leaf_fixed` ; soit la taille est donnée par un autre champ, dans ce cas, `t_path` donne le champ associé ;

```
type t_gfm =
| Gfm_leaf of t_path
| Gfm_leaf_fixed of nat
| Gfm_len of t_path * t_gfm
| Gfm_len_fixed of nat * t_gfm
| Gfm_delim of t_path * t_gfm
| Gfm_delim_fixed of bytes * t_gfm
| Gfm_sequence of t_gfm * t_gfm
| Gfm_repetition of t_path * t_gfm
| Gfm_star of t_path * t_gfm
| Gfm_choice of t_path * nat * t_gfm * t_gfm
| Gfm_fct of fct
```

FIGURE 3.20 – Structure représentant un GFM pour Coq

soit le champ est délimité et le parent de ce champ est `Gfm_delim(_fixed)`. Le cas des délimiteurs est traité à part car de nombreuses obfuscations ne peuvent s'appliquer sur un nœud si un de ses parents est de type `Gfm_delim(_fixed)` (contraintes C.1 et C.2).

Comme les transformations qui modifient la taille du nœud possèdent une contrainte particulière supplémentaire (cf. tableau 3.3), nous avons classé les transformations en deux catégories :

— *TransLen* - les transformation ne modifiant pas la taille du nœud ;
— *TransTree* - les transformations modifiant la taille du nœud.

		Catégorie	
		Agrégation	Ordonnancement
Famille	TransLen	SplitCat **ConstAdd** ConstSub ConstXor	**AddLen** **ReadFromEnd** DissectFromEnd ChildMove **RepSplit**
	TransTree	**SplitAdd** SplitSub SplitXor DupNode TreeCrypto	**PadInsert** BoundaryChange StarSplit

TABLE 3.4 – Liste des transformations par famille et catégorie, en gras les transformations qui ont été prouvées avec Coq.

L'exemple donné en début de chapitre (section 3.1.2) peut aussi être représenté suivant le modèle Coq. La figure 3.21 montre que la représentation reste compacte car les champs qui auraient besoin d'un nœud `Gfm_len(_fixed)` en parent sont généralement les leafs, et avec ce modèle, la relation peut être directement écrite.

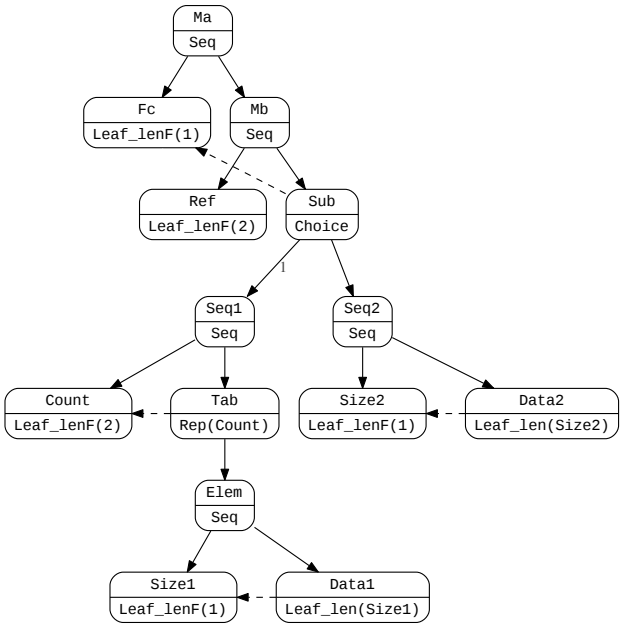

FIGURE 3.21 – Exemple de graphe de format de message comportant deux messages différenciés par le mot clé *Fc*. Cette représentation correspond au modèle Coq.

3.3.1 Coq : validation du `parser` et du `serializer`

Avec le soutien du programme Coq, nous avons établi les relations de la figure 3.22. Dans cette notation, `Some` signifie que l'évaluation d'une fonction a réussi. La valeur `None` signifie que cette évaluation a echoué. De plus, l'opérateur `"++"` correspond à une concaténation de chaînes de caractères. La fonction `parse` transforme un message en AST tandis que la fonction `serialize` transforme un AST en message. Ainsi, le typage de la fonction `parse` est : `parse: mem -> gfm -> msg -> (ast * msg * rem) option`. Autrement dit, cette fonction accèpte 3 paramètres (une mémoire de parcours, un gfm et un message) et retourne `None` en cas d'échec, ou l'ast obtenu par le parseur, empaqueté dans la valeur `Some`. Le typage de la fonction `serialize` est : `serialize: mem -> gfm -> ast -> msg option`. Autrement dit, cette fonction accepte 3 paramètres (une mémoire de parcours, un gfm et un message) et retourne `None` en cas d'échec, ou le message obtenu par le serialiseur, empaqueté dans la valeur `Some`. Le détail de ces fonctions est donné en section 3.1.3.

Un message invalide est un message qui ne peut pas être parsé correctement pour un gfm donné (i.e., le résultat de la fonction `parse` est `None` pour ce message). De la même manière, un ast invalide est un ast qui ne peut pas être sérialisé correctement pour un gfm donné (i.e., le résultat de la fonction `serialize` est `None` pour cet ast). Cette notion d'invalidité est fortement basée sur la manière dont le

parseur reconstruit l'ast et dont le sérialiseur enchaîne la traduction des nœuds en séquence d'octets. Durant ces traductions, des portions de message (pour le parseur) et des nœuds d'ast (pour le sérialiseur) sont confrontés au gfm. Par exemple, si le nœud de l'ast à sérialiser ne correspond pas à ce qui est attendu au niveau du gfm, alors le sérialiseur stoppe la traduction et renvoie `None`. Par conséquent, il est important de considérer les deux derniers lemmes de la figure 3.22 qui traitent des cas d'erreur `None`. Le lemme *serialize_none_parse_none* assure qu'un message invalide ne peut pas avoir été généré par la sérialisation d'un AST, et le lemme *parse_none_serialize_none* qu'un AST invalide ne peut pas être engendré par le parsing d'un message.

`Lemme` *serialize_parse_id* :
∀ *gfm mem ast msg' rem*,
`(*S'il est possible de transformer un ast en message msg', ...*)`
serialize mem gfm ast = *Some msg'* →
`(*... alors il doit être possible de parser un message avec msg' en préfixe, pour obtenir à nouveau l'ast initial.*)`
parse mem gfm (*msg'* ++ *rem*) = *Some* (*ast*, *msg'*, *rem*).

`Lemme` *parse_serialize_id* :
∀ *gfm mem msg ast msg' rem*,
`(*S'il est possible de parser un msg pour obtenir un ast, ...*)`
parse mem gfm msg = *Some* (*ast*, *msg'*, *rem*) →
`(*... alors la sérialisation de cet ast doit aboutir au message lui-même.*)`
serialize mem gfm ast = *Some msg'*.

`Lemme` *serialize_none_parse_none* :
∀ *gfm mem ast msg' rem*,
`(*Si un ast ne peut pas être sérialisé, ...*)`
serialize mem gfm ast = *None* →
`(*... alors il ne doit pas pouvoir être obtenu par le parsing d'un message.*)`
parse mem gfm (*msg'* ++ *rem*) ≠ *Some* (*ast*, *msg'*, *rem*).

`Lemme` *parse_none_serialize_none* :
∀ *gfm mem ast msg' rem*,
`(*Si un message ne peut pas être parsé, ...*)`
parse mem gfm (*msg'* ++ *rem*) = *None* →
`(*... alors ce message ne doit pas pouvoir être obtenu par la sérialisation d'un ast.*)`
serialize mem gfm ast ≠ *Some msg'*.

FIGURE 3.22 – Lemmes associés aux fonctions parse et serialize

Le premier lemme *serialize_parse_id* assure que tout message généré à par-

tir d'un AST valide (vis-à-vis d'un GFM), peut être correctement parsé (avec ce même GFM) et aboutit au même AST, i.e. $parse(serialise) = id$. Le second lemme *parse_serialize_id* assure que tout AST parsé correctement à partir d'un message (vis-à-vis d'un GFM) génère le même message, i.e. $serialise(parse) = id$. Le lemme *serialize_none_parse_none* assure également qu'un message invalide ne peut pas avoir été généré par un AST, et le lemme *parse_none_serialize_none* qu'un AST invalide ne peut pas être obtenu par le parsing d'un message. Les deux derniers lemmes sont des corollaires des deux premiers et garantissent que la fonction serialize est l'inverse de la fonction parse et inversement : $parse \circ serialize = id$. et $serialize \circ parse = id$ (en employant une écriture abusive étant donné que les prototypes des fonctions ne coïncident pas).

3.3.2 Coq : validation des obfuscations

Une obfuscation (τ : `t_gfm` $\rightarrow$ `option t_gfm`) transforme un GFM gfm en un autre GFM gfm' doit préserver la validité des ASTs (il ne faut pas changer le langage manipulé par l'application qui utilise la grammaire). La validité d'un AST reposant sur le résultat de la fonction `serialize`, nous en déduisons les deux lemmes de la figure 3.23. Notons que les lemmes sur `serialize` suffisent et que les lemmes sur parsing sont obtenus en appliquant l'idempotence à deux reprises, en combinaison des lemmes de figure 3.22.

```
Lemme parse_serialize_trans1 :
  ∀ trans gfm gfm' ast mem msg,
  (*Si l'application d'une transformation est réussie, ...*)
    trans gfm = Some gfm' →
    (*... et qu'un ast est valide vis-à-vis du gfm initial, ...*)
      serialize mem gfm ast = Some msg →
      (*... alors cet ast doit être valide pour le gfm obfusqué.*)
        ∃ msg', serialize mem gfm' ast = Some msg'.

Lemme parse_serialize_trans2 :
  ∀ trans gfm gfm' ast mem msg',
  (*Si l'application d'une transformation est réussie, ...*)
    trans gfm = Some gfm' →
    (*... et qu'un ast est valide vis-à-vis du gfm obfusqué, ...*)
      serialize mem gfm' ast = Some msg' →
      (*... alors cet ast doit être valide pour le gfm initial.*)
        ∃ msg, serialize mem gfm ast = Some msg.
```

FIGURE 3.23 – Lemmes associés aux transformations pour garantir leur inversibilité

Les transformations en gras dans le tableau 3.4 correspondent aux transformations pour lesquelles nous avons validé l'inversibilité avec l'outil coq. Les autres transformations sont soit plus simples soit similaires et devraient suivre le même raisonnement pour leur validation.

3.4 Conclusion

Dans ce chapitre, nous avons présenté une nouvelle famille d'obfuscations de protocoles. Cette famille s'appuie sur la modification du format du message pour créer des protocoles au format plus complexe. Nous proposons plusieurs obfuscations unitaires qui sont composables les unes avec les autres. Cela permet de plus simplement implémenter ces obfuscations (en chapitre 4) ainsi que de travailler à prouver leur réversibilité. Nous classons ces obfuscations d'après [Collberg 1997] : transformations d'agrégation ou d'ordonnancement. À chaque obfuscation, nous associons des contraintes qui doivent être respectées pour pouvoir l'appliquer. Certaines contraintes sont générales à toutes les obfuscations que nous proposons et d'autres sont spécifiques à certaines obfuscations. Ces obfuscations s'appuient sur certaines hypothèses de la rétro-conception afin de la rendre plus difficile. Ainsi, les modèles créés par ces obfuscations peuvent ne plus être réguliers, les identifiants de messages peuvent disparaître, les délimiteurs sont enlevés. Les tableaux 3.1 et 3.2 donnent pour chaque obfuscation les challenges ou hypothèses de la rétro-conception sur lesquels elle s'appuie. Des travaux sont en cours afin de valider formellement ces obfuscations avec l'appui de l'outil Coq. Certaines ont d'ores et déjà été validées.

L'implémentation, quant à elle, s'appuie sur la classification de Collberg *et al.*. Si le parseur et le serialiseur proposés dans ce chapitre sont génériques et prennent en paramètre le GFM, nous avons développé un prototype permettant de générer un parseur et un serialiseur spécifiques à un GFM. Ainsi, ces codes intègrent directement les modifications apportées au GFM par les obfuscations. Si l'on considére un attaquant pratiquant la rétro-conception, cette approche est plus intéressante car il est généralement plus compliqué de pratiquer la rétro-conception de plusieurs fonctions, plutôt que la rétro-conception d'une seul fonction ayant en paramètre le schéma de parcours. Les problématiques liées à l'implémentation sont développées dans le prochain chapitre. La robustesse de ces obfuscations repose sur la capacité de composition de ces transformations. Ainsi, une évaluation du prototype en fonction de la profondeur de composition est proposée dans le chapitre 5.

Chapitre 4

Implémentation de *ProtoObf*

Sommaire

Ce chapitre présente un framework permettant de mettre en œuvre les obfuscations de spécifications de protocoles, telles que présentées dans le chapitre précédent. Nous le nommons *ProtoObf*. L'objectif de ce framework est double. Il permet tout d'abord de valider le modèle des obfuscations par une implémentation concrète. Le second objectif de ce framework est de fournir un environnement de développement à des utilisateurs/développeurs facilitant la mise en place automatique d'obfuscations sur leur protocole. Autrement dit, ce framework permet aux développeurs, une fois leur code écrit, d'appliquer automatiquement et très simplement des obfuscations différentes sur le format des messages de leur protocole, sans nécessiter aucune modification de leur code.

4.1 Architecture de *ProtoObf*

L'idée d'un tel framework est de générer automatiquement le code qui gère la sérialisation et le parsing des données. La structure de données échangées est définie en amont, elle correspond au format des messages du protocole. Elle est remplie à partir de *getters* et de *setters* appelés *accesseurs*. L'utilisation des accesseurs permet de résoudre les obfuscations d'agrégation qui modifient la manière dont les données sont agrégées. Le code de sérialisation contient quant à lui les obfuscations d'ordonnancement et le code de parsing contient les obfuscations inverses. Tant que l'interface d'utilisation des fonctions de sérialisation, de parsing et des accesseurs ne change pas, l'utilisateur n'a pas à changer son code pour intégrer les obfuscations sur le format du message. L'utilisation d'accesseurs permet notamment de faire en sorte que le message en clair n'est jamais présent en mémoire en totalité.

Il existe différents outils de sérialisation des données tels que Protobuff [Google 2017], Cap'n'Proto [Varda 2017], Thrift [Apache 2017], RPC Windows [Microsoft Corporation 2018] ou ASN.1 [asn 2017]. Ces outils permettent de plus l'utilisation de *Remote Procedure Call* (*RPC*). De part leur finalité complexe (les RPC impliquent de savoir faire de la sérialisation de données), l'intégration de notre technologie à ces projets existants demande un fort travail d'ingénierie. De plus, comme nous le montrons en section 4.2.2, la composition des obfuscations nous amène à modifier la représentation interne qui est classiquement utilisée par ces outils. Ainsi, nous avons choisi de développer un nouveau prototype.

La structure des messages est spécifiée dans un *DSL* (*Domain Specific Language*). Cette spécification est parsée et traduite vers une représentation interne qui sert effectivement pour la génération du code de sérialisation et de parsing des messages. Le développeur peut alors utiliser cette bibliothèque générée, via une interface de développement spécifique. Les données sont sauvegardées dans une autre représentation interne (pour le code généré) équivalente à la représentation interne du DSL (pour l'outil de génération de code). La manipulation de ces données se fait via des accesseurs.

En gardant une représentation interne de la spécification du message équivalente à un GFM présenté dans le chapitre 3, il est possible d'appliquer des obfuscations avant la phase de génération de code. Alors, il suffit de modifier légèrement le générateur de code pour prendre en compte les nouveaux types de nœuds correspondant aux obfuscations. Ainsi, la figure 4.1 représente cette architecture modifiée pour prendre en compte les obfuscations. Cette architecture est présentée pour la première fois dans [Duchêne 2016b]. Le framework avec l'implémentation des obfuscations est ensuite détaillé dans [Duchêne 2018].

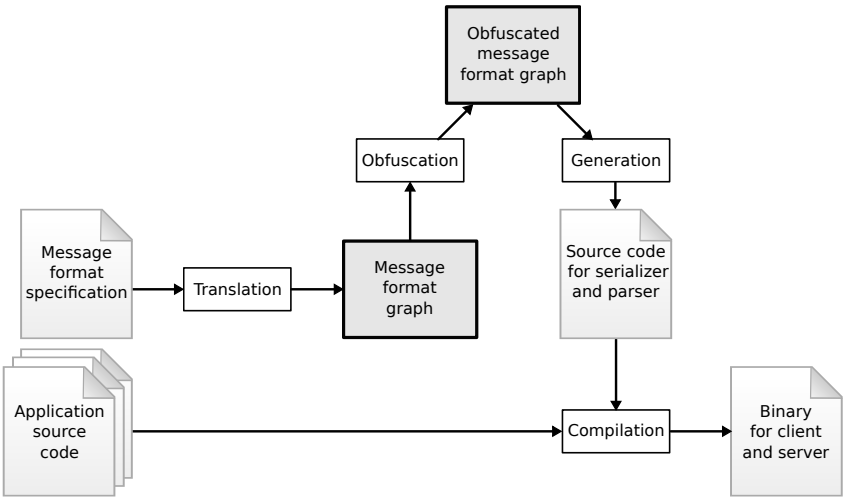

FIGURE 4.1 – Architecture de protoobf.

Cependant, il est important de faire attention à conserver la même API (Application Programming Interface) pour que l'utilisateur n'ait pas à modifier son code

lors de l'utilisation d'obfuscations. Ainsi, la figure 4.2 explicite les éléments (en rouge) qui changent à chaque compilation et ceux qui restent fixes (en bleu). Dans le code généré, l'interface de programmation reste inchangée. De plus, il faut s'assurer que l'application automatique des obfuscations est bien intégrée à la chaîne de compilation. Ce processus est détaillé en section 4.2.4 grâce à l'utilisation de CMake.

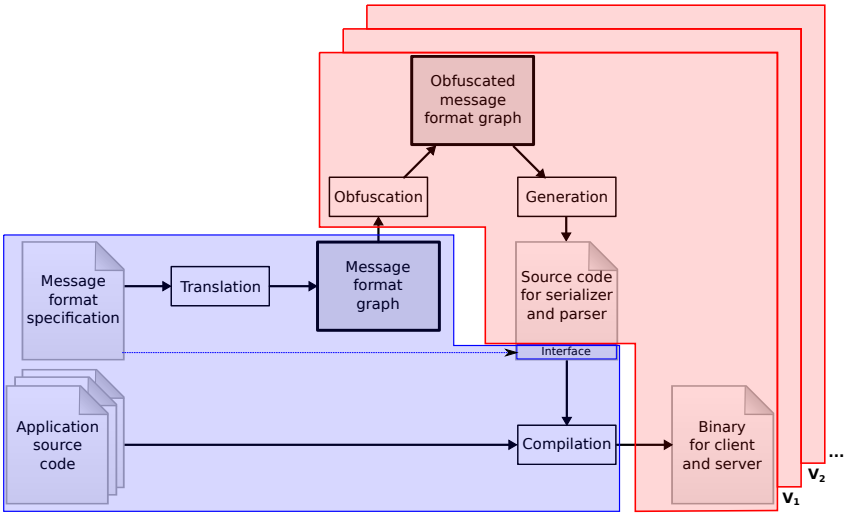

FIGURE 4.2 – Élements non modifiés (en bleu) ou modifiés (en rouge) à chaque compilation d'un projet basé sur ProtoObf.

4.2 Implémentation

Dans cette section, nous détaillons les choix d'implémentation pour la réalisation du framework. Le but est d'obtenir un outil :

- intégrable à une chaîne de compilation classique ;
- appliquant automatiquement des obfuscations sur le format de message ;
- changeant les obfuscations à chaque compilation ;
- gardant les mêmes interfaces de parsing, sérialisation et d'accesseurs sur les messages.

Pour cela, nous avons défini notre propre DSL qui est manipulé avec Flex/Bison [1]. Les éléments sont ensuite convertis dans une représentation interne à base de bi-graphes (détaillés en section 4.2.2). L'algorithme de choix des obfuscations est implémenté en C. Le code de la librairie de parsing, sérialisation et accesseurs associé à la spécification est généré à partir de modèles de codes développés en C.

1. `langevin.univ-tln.fr/CDE/LEXYACC/Lex-Yacc.html`

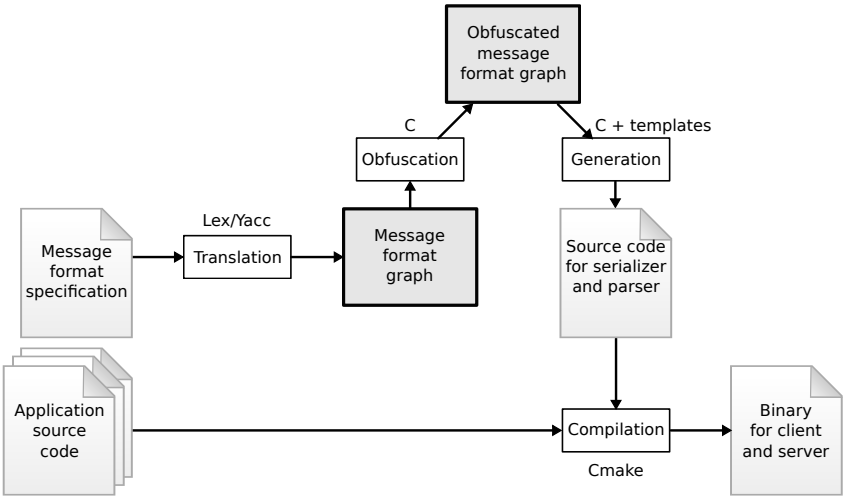

FIGURE 4.3 – Architecture de protoobf avec les choix d'implementation pour les différents modules.

4.2.1 Spécification du format de message : *DSL*

Nous avons adopté la création d'un nouveau *DSL* (*Domain Specific Language*) pour le format des messages car nous souhaitions pouvoir définir explicitement l'encodage (définir les bornes des champs) des messages dans la spécification. Cela nous a simplifié le travail de validation fonctionnelle de l'implémentation.

Dans notre DSL, pour faciliter la validation du prototype, nous avons choisi de spécialiser les `Gfm_leaf` (que l'on notera aussi `Leaf`) en fonction du type de données. Ainsi, une `Leaf` peut être : une donnée utilisateur, notée `Leaf:data` ; un mot-clé du protocole, notée `Leaf:key` ; la taille d'un nœud, notée `Leaf:len` ; le compteur d'une `Gfm_repetition`, noté `Leaf:count`. Cette spécialisation des `Leaf` a aussi été décidée pour se rapprocher des autres projets existants. Dans cette version du framework, les dissections à base de `Gfm_delim` (*i.e.* des délimiteurs donnés par un autre nœud) sont permis mais rapidement traduits en `Gfm_len` pour contourner les contraintes d'application des obfuscations. De plus, toujours par souci de convergence avec les projets existants, les `Gfm_sequence` peuvent contenir plus de deux éléments.

De plus, ce DSL nous permet de spécifier une plus large variété de protocoles que la plupart des projets existants. Il permet notamment de représenter des protocoles tels que HTTP ce qui n'est pas possible avec les frameworks cités précédemment. En effet, ces projets ne permettent pas l'utilisation de dissecteurs `Gfm_delim(_fixed)`. Cela s'explique par une meilleure efficacité des parseurs lors de l'utilisation de `Gfm_len(fixed)` ainsi qu'une réduction des contraintes sur les champs (séquence délimiteur interdite dans le contenu du champ).

Nous n'avons pas utilisé ASN.1 [asn 2017] car il était trop complexe et nous avons préféré une spécialisation de notre DSL pour pouvoir effectuer plus simplement des obfuscations sur le format des messages. Ainsi, nous avons utilisé et étendu le DSL utilisé par les projets Protobuff [Google 2017], Cap'n'Proto [Varda 2017] et

Windows RPC [Microsoft Corporation 2018]. Il est facilement possible d'exprimer un message dont le format est spécifié dans le DSL de Protobuff ou de Cap'n'Proto dans notre propre DSL. Ces frameworks permettent de définir les types suivants :

— *Donnée utilisateur* (`Leaf:data`) - un buffer arbitraire avec une taille ;
— *Mot-clé* (`Leaf:key`) - un élément d'une énumération ;
— *Séquence* (`Gfm_sequence` aussi noté `Seq`) - une séquence de sous-champs ;
— *Tableaux* (`Gfm_repetition` aussi noté `Rep`) - un tableau de sous-champs ;
— *Listes* (`Gfm_star` aussi noté `Star`) - une répétition de sous-champs.

Contrairement aux précédents DSL, nous avons choisi de pouvoir expliciter les champs servant à l'encodage des données. Ce sont des champs terminaux au même titre que les nœuds `Leaf:key` ou `Leaf:data`. Nous avons ainsi ajouté les types suivants :

— *Compteur* (`Leaf:count`) - un compteur donnant la taille d'un tableau ;
— *Taille* (`Leaf:len`) - la taille d'un autre champ.

Ces champs existaient de manière implicite dans les précédents frameworks, mais ils n'étaient pas contrôlables par l'utilisateur.

De plus, pour pouvoir représenter les messages de protocoles tels que `HTTP`, nous avons besoin d'introduire les champs optionnels. Nous avons donc ajouté le type de nœud *Optionnel* (`Gfm_choice` aussi noté `Choice`), la branche de l'option (sous-graphe) est déterminée par la valeur d'un nœud `Leaf:key`. De plus, dans ce DSL, le développeur doit spécifier les bornes de chaque champ, *i.e.* comment est disséqué chaque champ, ce qui correspond à spécifier le `t_dissect` des champs associés.

Les messages sont décrits sous forme de règles reprenant le principe de la notation *BNF* à laquelle a été ajoutée la dissection et le typage des champs. Ainsi le format d'un champ se représente sous la forme :

> *nom_du_champ* := [*dissecteur*] : *type_de_regle* : *valeur* ou liste de *nom_de_champ*

Tout message qui est écrit dans ce DSL doit aussi respecter les contraintes du modèle présenté en chapitre 3. En effet, des vérifications sont effectuées lors du parsing du format du message dans le DSL avec Flex/Bison. Cependant, cette vérification n'est pas encore complètement implémentée. Pour le moment, nous supposons que le message décrit est correct. Les différences entre les mots-clé du DSL et ceux du modèle sont considérées comme mineures, voici le lien entre ces deux nommages. Pour le dissecteur :

— *Fixed* correspond à `Gfm_len_fixed`, la taille étant donnée juste après ;
— *Length* correspond à `Gfm_len`, le `t_path` est donné par le nom du nœud entre {} qui suit ;
— *Delim* correspond à `Gfm_delim_fixed`, le délimiteur étant donné juste après entre guillemets ;
— *Child* correspond à l'absence de `t_dissect`.

De plus, pour le type de règle :

— *E* correspond à une `Leaf` dont la spécialisation est donnée juste après (`Data`, `Keyword`, `Counter` ou `Length`) ;

— T correspond à un tableau, soit `Gfm_repetition` ;
— O correspond à une option, soit `Gfm_choice` ;
— L correspond à une séquence, soit `Gfm_sequence` ;
— R correspond à une liste, soit `Gfm_star`.

Exemple : Le message M (liste 4.1) est composé de trois sous-champs : Fc, $Count$ et $Coils$. $Coils$ est un tableau dont la taille est donnée par le champ $Count$, et contient un sous-champ $Coil$ de type `Séquence`. $Coil$ est composé de deux sous champs S et D.

Listing 4.1 – Exemple de message écrit dans le DSL

```
M := Fc Count Coils
Fc := [Fixed,1] : E : Keyword,10
Count := [Fixed,1] : E : Counter
Coils := [Child] : T : Coil{Count}
Coil := [Child] : L : S D
S := [Fixed,1] : E : Length
D := [Length,{S}] : E : Data
```

Limitations du DSL Chaque champ ne peut avoir qu'un seul type, ainsi, il n'est pas possible par exemple de représenter une grammaire où un champ serait à la fois la taille d'un tableau et la taille d'un autre champ. De plus, les cycles sont interdits et les champs sont identifiés par leur nom de manière unique.

4.2.2 Implémentation des obfuscations

Les obfuscations d'ordonnancement sont intégrées dans les fonctions de parsing/sérialisation tandis que les obfuscations d'agrégation sont contenues dans les accesseurs. Les accesseurs doivent préserver l'API initiale pour les messages obfusqués. Effectivement, un développeur doit pouvoir intégrer le framework dans sa chaîne de développement sans avoir à modifier son code. Ainsi, le GFM manipulé par le framework pour générer le code (*parseur*, *sérialiseur* et *accesseurs*) doit donc garder la structure du GFM d'origine (pour les *accesseurs*) en mémoire afin de préserver la cohérence des accesseurs. Ainsi, il est naturel de séparer la partie du modèle qui doit être préservée de celle qui peut librement évoluer. Nous avons choisi d'utiliser deux graphes que nous nommons *bi-graphe*. Le premier est utilisé pour les accesseurs et est appelé *graphe d'agrégation*. Le second sert pour le parseur et le sérialiseur, il est appelé *graphe d'ordonnancement*.

Ainsi, le *graphe d'agrégation* permet de construire le code des accesseurs en gardant la structure d'origine du graphe et ajoute des nœuds d'obfuscation correspondant aux agrégations à effectuer pour retrouver le contenu des nœuds avant obfuscation. Il permet donc de générer une structure interne qui contient les données effectivement sérialisées ou parsées, *i.e.* les données après obfuscations, ainsi

que les fonctions pour reconstruire les données non-obfusquées. Une obfuscation d'agrégation modifie donc principalement le graphe d'agrégation, mais si des nœuds terminaux (*i.e.* les données parsées ou sérialisées) sont modifiés, alors le graphe d'ordonnancement est modifié en conséquence. Ce graphe ne contient pas d'informations de dissection, seulement de typage des données.

Le *graphe d'ordonnancement* est utilisé pour générer le code du parseur et du sérialiseur, *i.e.* la sélection des données qui seront effectivement envoyées sur le réseau et leur ordonnancement. Comme le nœud racine ne change pas, l'interface de parsing et de sérialisation est automatiquement préservée. En effet, le développeur n'appelle que la fonction de sérialisation ou de parsing racine. Ce graphe peut être applati, *i.e.*, il est possible d'enlever les nœuds de type `Gfm_sequence`, seule la concaténation des `Gfm_leaf` est requise. Cependant, les nœuds de type `Gfm_repetition`, `Gfm_star` et `Gfm_choice` nécessitent un traitement particulier qui ne peut être réduit.

La figure 4.4 donne un exemple de bi-graphe pour le message donné en liste 4.1. Nous avons créé un lien d'affectation entre le graphe d'ordonnancement (de droite) et le graphe d'agrégation (de gauche) pour indiquer où les données doivent être lues lors de la sérialisation ou écrites lors du parsing. En pratique un nœud terminal du graphe d'ordonnancement possède un pointeur vers le nœud du graphe d'agrégation associé. Cela permet lors de la génération du code du paseur ou du sérialiseur de générer le code qui retrouve dans la structure interne le bon nœud à écrire où à lire. De plus, les liens de dépendances (`t_path`) sont créés entre le graphe d'ordonnancement et le graphe d'agrégation (les flèches en pointillées sur la figure 4.4 et un pointeur dans le code). Cette représentation interne (bi-graphe) ne se retrouve pas dans le code généré. Les figures 4.4 à 4.6 montrent l'enchaînement de l'application de 2 obfuscations : *SplitOp* sur S ; puis *RepSplit* sur $Coils$.

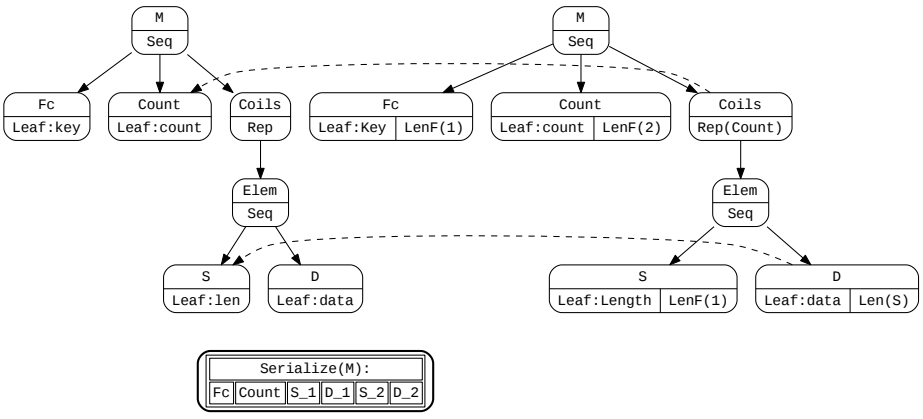

FIGURE 4.4 – Bi-graphe initial du message M, en dessous est figurée une sérialisation en supposant que le tableau contient 2 éléments.

Encore une fois, nous avons choisi cette solution car nous la trouvons élégante

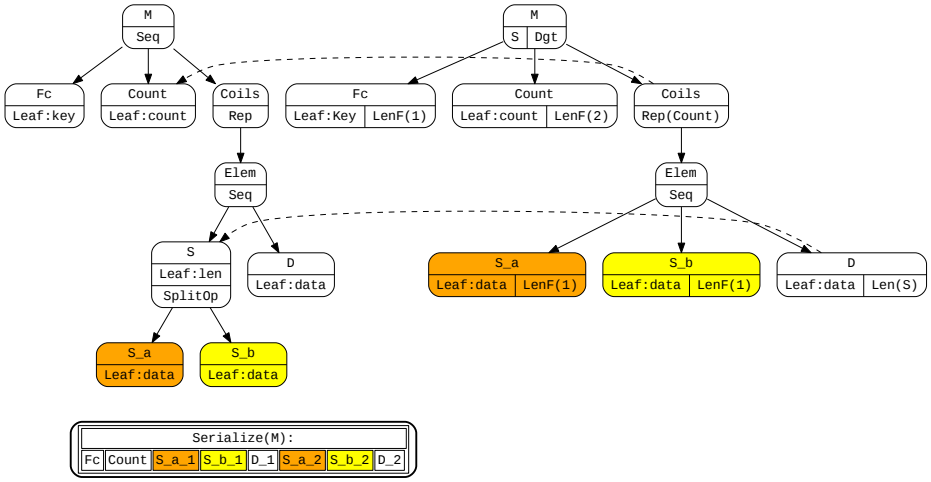

FIGURE 4.5 – Bi-graphe du message M avec application de `SplitOp` sur le nœud S. Les couleurs correspondent aux nœuds ayant subi une transformation.

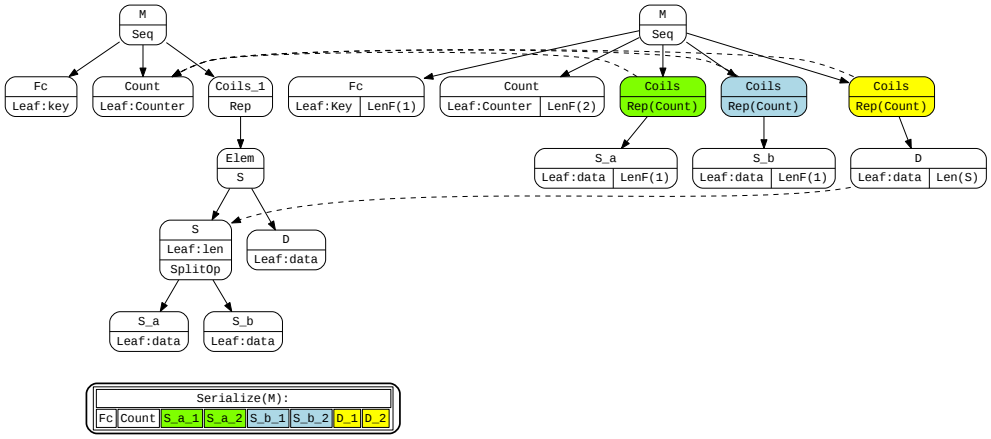

FIGURE 4.6 – Bi-graphe du message M avec application de `SplitOp` sur le nœud S, puis de `RepSplit` sur *Coils*. Les couleurs correspondent aux nœuds ayant subi la dernière transformation.

et qu'elle nous simplifie l'implémentation. Nous aurions pu utiliser un seul graphe. Or, dans ce cas, il faut garder en mémoire quels sont les nœuds d'origine pour la cohérence des interfaces de développement générées. De plus, l'ensemble des obfuscations doit être sauvegardé (et non plus seulement les obfuscations d'agrégation) dans ce graphe car il doit assurer à la fois la cohérence du parseur/sérialiseur et des accesseurs. Nous trouvons ce graphe plus difficile à manipuler et à maintenir. Un exemple est donné en figures 4.7 à 4.9 où le même enchaînement d'obfuscations

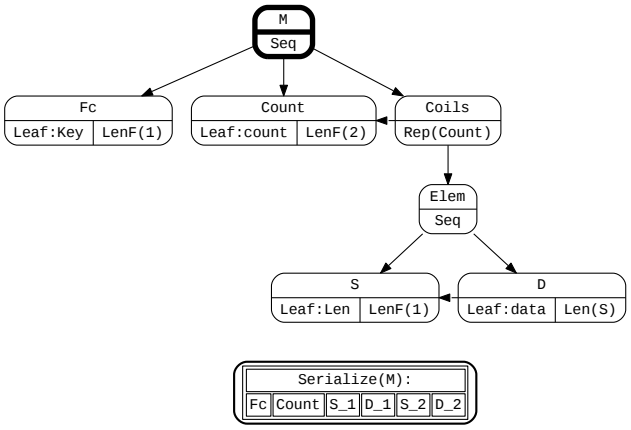

FIGURE 4.7 – GFM initial d'un message M, en dessous est figuré une sérialisation en supposant que le tableau contient 2 éléments.

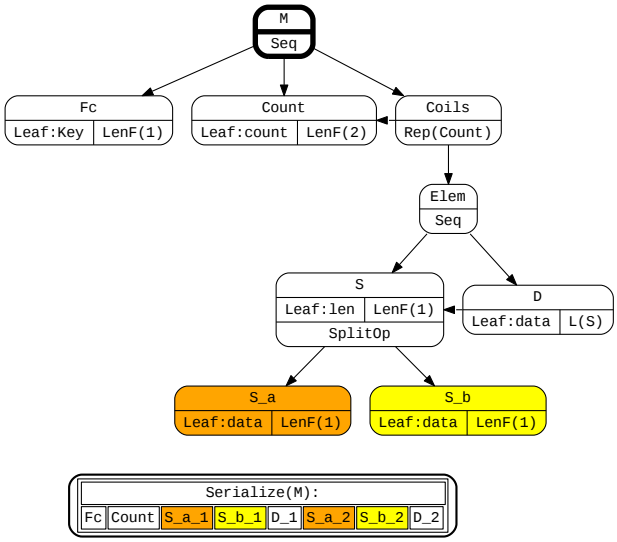

FIGURE 4.8 – GFM du message M avec l'obfuscation `SplitOp` sur le nœud S. Le nœud D dépend en taille de S. Les couleurs correspondent aux nœuds ayant subi une transformation.

SplitOp sur S et *RepSplit* sur $Coils$ est effectué. Nous n'avons pas fait figurer sur ces figures la sauvegarde de la notion qu'un nœud est un nœud d'origine. Trois inconvénients sont à noter sur la figure 4.9 par rapport au cas de la figure 4.6. Les nœuds $Coils$ et S sont présents dans le graphe final, et seront donc utilisés pour la génération du parseur et du sérialiseur. Dans le cas du *bi-graphe*, ils ont pu être

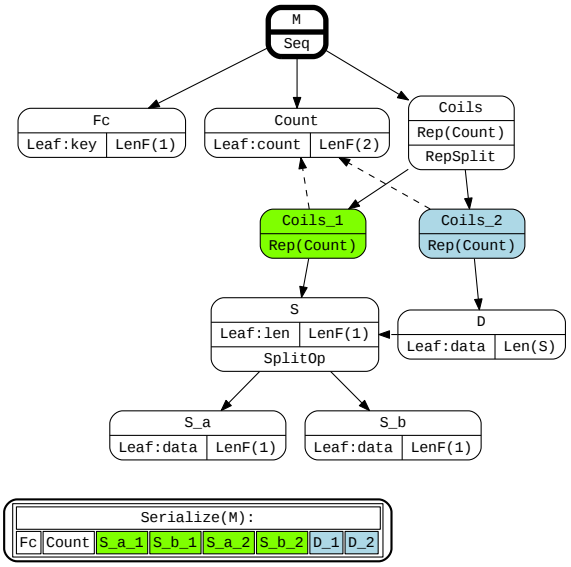

FIGURE 4.9 – GFM du message M avec la composition d'obfuscations : *SplitOp* sur le nœud S et `RepSplit` sur $Coils$. Le nœud $Coils$ est toujours présent. Les couleurs correspondent aux nœuds ayant subi une transformation.

supprimés. Le second inconvénient est que le tableau *Coils_1* ne peut pas être de nouveau découpé. En effet, le nœud S doit être conservé pour le parsing de D. Or, une séquence de 3 tableaux est généralement plus complexe à inférer qu'une séquence de 2. Le dernier inconvénient, qui n'a pas été représenté sur la figure pour la lisibilité justement, est que l'on doit garder la trace des nœuds originaux pour pouvoir générer les même interfaces d'accesseurs pour le graphe d'origine et pour le graphe obfusqué.

4.2.3 Génération de code par des templates

Le code généré par le framework repose sur le principe de templates de codes où en parcourant le graphe idoine (d'ordonnancement pour le code du parseur et du sérialiseur, d'agrégation pour les accesseurs), le code de la fonction dépend du type de nœud. Ainsi, voici les étapes associées à chaque type de nœud :

— `Gfm_repetition`, la taille du tableau (`Leaf:count`) est lue, un code de parcours de tableau est généré ; dans la boucle de parcours du tableau, un appel à la fonction associée au sous nœud est réalisé.
— `Gfm_star`, un code de parcours de liste chaînée est généré ; dans la boucle de parcours de la liste, un appel à la fonction associée au sous nœud est réalisé.
— `Gfm_sequence`, un appel à la fonction associée à chaque sous nœud est réalisé.
— `Gfm_choice`, une vérification du `Leaf:key` (représentant le choix) est effectuée, un appel à la fonction associée au sous-nœud correspondant au keyword

choisi est réalisé.
— `GFM_leaf`, fonction effectuant les tâches suivantes : lecture du buffer et sauvegarde dans la représentation interne, ou lecture dans la représentation interne et écriture dans le buffer.

De plus, pour les fonctions `Parse`, le code associé au dissecteur est généré.

4.2.3.1 `Prototypes des fonctions`

Les données sont manipulées à travers des fonctions exportées dont le prototype reste indépendant des obfuscations envisagées. De plus, il y a des fonctions internes de manipulation des données qui prennent en compte les obfuscations mais qui ne sont pas exportées. Pour chaque champ d'un message, 9 fonctions sont générées :

— `Alloc` - alloue une structure interne permettant de représenter le champ
— `Free` - libère la structure interne et tout ce qui a été affecté dans cette structure
— `Parse` - parse un buffer de données brutes et retourne une structure interne pour la manipulation du champ (seule la fonction au niveau du message est exportée)
— `GetSize` - calcule la taille nécessaire pour serialiser un champ (seule la fonction au niveau du message est exportée)
— `Serialize` - transforme la structure interne en un buffer de données brutes sérialisées (seule la fonction au niveau du message est exportée)
— `Get` - renvoie la valeur des données d'un champ
— `Set` - affecte la valeur des données d'un champ
— `Initialize` - appelé avant le début de la sérialisation (fonction interne uniquement)
— `Finalize` - appelé après la fin du parsing ou lors du parsing en cas de dépendance sur des données précédentes (fonction interne uniquement)

Les fonctions Allocate, Free et les accesseurs peuvent être appelés par l'utilisateur final, mais les fonctions Serialize, Parse et GetSize doivent être appelées uniquement au niveau du message, les fonctions appliquées aux champs sont utilisées pour les manipulations internes.

Les accesseurs sont spécialisés en fonction du type de champ :

— `GFM_leaf` : accesseur simple pour lire/écrire le contenu d'un champ
— `GFM_sequence` : accesseurs simples pour lire/écrire le sous-champ d'un champ
— `GFM_repetition` : accesseurs pour lire/écrire directement sur chaque élément du tableau ainsi qu'une fonction pour affecter la taille du tableau (si le champ `Counter` a déjà une valeur qui est différente, lever une alerte).
— `GFM_star` : accesseurs pour ajouter un élément à la liste ainsi que pour parcourir la liste (GetFirst, GetNext).
— `GFM_choice` : identique à `Child` avec en plus la vérification de la valeur du `Leaf:key`.

A titre d'exemple, nous listons ci-dessous le prototype d'une partie des fonctions exportées pour manipuler les données du message M.

```
bool Allocate_Message_M(Message_M_T** ppMessage);
bool Free_Message_M(Message_M_T* pMessage);
bool Parse_Message_M (Message_M_T** ppMessage, uint8_t* pData,
↪  uint32_t ui32DataSize);
bool GetSize_Message_M (Message_M_T* pMessage, uint32_t*
↪  pui32Size);
bool Serialize_Message_M (Message_M_T* pMessage, uint8_t* pData,
↪  uint32_t* pui32DataSize);
bool Initialize_Message_M (Message_M_T* pMessage);
bool Finalize_Message_M (Message_M_T* pMessage);

bool Message_M_get_Field_M_Fc (Message_M_T* pMessage,
↪  Field_M_Fc_T** ppField);
bool Message_M_set_Field_M_Fc (Message_M_T* pMessage, Field_M_Fc_T*
↪  pField);
...
```

4.2.3.2 Représentation des champs dans le code généré

L'utilisateur final utilise des accesseurs pour accéder aux différents champs. Le code généré utilise des structures différentes de celles utilisées pour la génération du code. Pour chaque champ est créé une structure différente dont le contenu dépend du type de champ.

`GFM_sequence` : la structure générée aura autant de pointeurs vers des sous-structures associées à ses sous-GFM ;

`GFM_repetition` : la structure générée aura un pointeur vers un tableau dont la sous-structure est associée à son sous-GFM ;

`GFM_choice` : la structure générée aura une union de pointeurs vers des sous-structures associées à ses sous-GFM ;

`GFM_star` : la structure générée aura une liste chainée avec une référence vers la sous-structure associée à son sous-GFM ;

`GFM_leaf` : la structure générée aura une donnée et sa taille, la donnée est typée respectivement avec le sous-type de `Leaf:X`.

A titre d'exemple, le code qui sera généré pour le stockage des données pour le message M est le suivant.

```
typedef struct Message_M_T_ {
  struct Field_M_Fc_T_*    pField_M_Fc;
  struct Field_M_Count_T_*  pField_M_Count;
  struct Field_M_Coils_T_*  pField_M_Coils;
} Message_M_T;
```

```
typedef struct Field_M_Fc_T_ {
  uint32_t  ui32ElementTypeKeywordValue;
  bool      bIsDefined;
} Field_M_Fc_T;

typedef struct Field_M_Count_T_ {
  uint32_t  ui32Counter;
  bool      bIsDefined;
} Field_M_Count_T;

typedef struct Field_M_Coils_T_ {
  struct Field_M_Coil_T_**  pTabFieldElement;
  uint32_t      ui32Index;
} Field_M_Coils_T;

typedef struct Field_M_Coil_T_ {
  struct Field_M_S_T_*  pField_M_S;
  struct Field_M_D_T_*  pField_M_D;
} Field_M_Coil_T;

typedef struct Field_M_S_T_ {
  uint32_t  ui32Length;
  bool      bIsDefined;
} Field_M_S_T;

typedef struct Field_M_D_T_ {
  uint8_t*  pData;
  uint32_t  ui32DataSize;
  bool      bIsDefined;
} Field_M_D_T;
```

Des accesseurs spéciaux sont créés pour la récupération et l'affectation des données depuis les fonctions de serialisation/parsing. Ces fonctions utilisent un contexte de parcours des données. Lors du parsing d'un tableau (`Gfm_repetition`) ou d'une liste (`Gfm_star`), les sous-fonctions ne savent pas sur quel élément du tableau ou de la liste elles travaillent : un index de parcours *ui32Index* est ajouté à la structure de tableau et un pointeur vers l'élément courant *pCurrentNodeField* est ajouté à la structure de liste. Ces accesseurs spéciaux permettent de retrouver les champs lors du parsing même lorsque des obfuscations ont été appliquées.

A titre d'exemple, les prototypes de quelques accesseurs spéciaux pour le message M sont les suivants. Le code complet des accesseurs spéciaux est présent en annexe C.

```
bool Message_M_get_Field_M_S (Message_M_T* pMessage, Field_M_S_T**
↪  ppField);
bool Message_M_set_Field_M_S (Message_M_T* pMessage, Field_M_S_T*
↪  pField);
bool Message_M_get_Field_M_D (Message_M_T* pMessage, Field_M_D_T*
↪  pField);
bool Message_M_set_Field_M_D (Message_M_T* pMessage, Field_M_D_T*
↪  pField);
```

4.2.3.3 Utilisation

Si l'on considère le message M (liste 4.1), voici un exemple de ce qu'un utilisateur devra faire pour générer les données à envoyer puis pour récupérer les données reçues. Il sait qu'il a deux données dans le tableau à envoyer. Dans cet exemple, le code ne contient pas toutes les vérifications nécessaires à un développement correct, telles que le test des codes de retour. Le code complet utilisé pour les évaluations du framework est disponible en annexe C.

Le code suivant donne un aperçu des différentes étapes à effectuer pour sérialiser un message M avant l'envoi.

```
// local variables
bool                res = false;
uint8_t             i;
Message_M_T*        pM;
Field_M_Fc_T*       pFc;
Field_M_Count_T*    pCount;
Field_M_Coils_T*    pCoils;
Field_M_Coil_T*     pCoil;
Field_M_S_T*        pS;
Field_M_D_T*        pD;
uint32_t            serializedSize;
uint8_t*            pSerializedBuffer;
uint8_t         ui8BufferSize = sizeof(buffer);
uint8_t       ui8TableSize = 2;

// allocate
Allocate_Message_M(&pM);
Allocate_Field_M_Fc(&pFc);
Allocate_Field_M_Count(&pCount);
Allocate_Field_M_Coils(&pCoils);

// set links
Message_M_set_Field_M_Fc(pM, pFc);
```

```
SetCounter_Field_M_Count(pM, pCount, ui8TableSize);
Message_M_set_Field_M_Count(pM, pCount);
Message_M_set_Field_M_Coils(pM, pCoils);

// work on the table
InitTab_Field_M_Coils(pM, pCoils);
for (i = 0; i < ui8TableSize; i++)
{
  Allocate_Field_M_Coil(&pCoil);
  Allocate_Field_M_S(&pS);
  Allocate_Field_M_D(&pD);

  Field_M_Coils_SetTabField_Field_M_Coil(pM, pCoils, pCoil, i);

  Field_M_Coil_Set_Field_M_S(pM, pCoil, pS);
  Field_M_Coil_Set_Field_M_D(pM, pCoil, pD);


  // prepare some random content to send (buffer)
  // buffer is feeded with content
  SetData_Field_M_D(pM, pD, buffer, ui8BufferSize);
}

GetSize_Message_M(pM, &serializedSize);

// allocate output buffer
MALLOC_ARRAY(serializedSize, pSerializedBuffer);
// serialize message
Serialize_Message_M (pM, pSerializedBuffer, &serializedSize);
// now can send data (pSerializedBuffer)
```

Le code suivant présente un aperçu des différentes étapes à effectuer pour parser un message M reçu afin d'accéder à ses différents composants.

```
// local variables
bool                res = false;
uint8_t             i = 0;
Message_M_T*        pM = NULL;
Field_M_Coils_T*    pCoils = NULL;
Field_M_Coil_T*     pCoil = NULL;
Field_M_D_T*        pD = NULL;
uint32_t            ui32MaxTabLen = 0;
uint32_t            ui32DataSize = 0;
// ui32Size and pData are known and initialized;
```

```
Parse_Message_M(&pM, pData, ui32Size);

// access to data to manipulate them :
GetCounter_Field_M_Count (pM, &pCoils);

GetTabLen_Field_M_Coils (pM, pCoils, &ui32MaxTabLen);

// treat each element of the table
for (i = 0; i < ui32MaxTabLen; i++)
{
  uint32_t  ui32DataSz = 0;
  uint8_t*  pData = NULL;

  // get field Coil
  Menc_T_GetTabField_Menc_E (pT, &pE, i, pCtx);
  Field_M_Coils_GetTabField_Field_M_Coil (pM, pCoils, &pCoil, i);

  // S can be ommited because it is used only for the parsing of D
  Field_M_Coil_Get_Field_M_D (pM, pCoil, &pD);

  // get values
  GetData_Field_M_D (pM, pD, NULL, &ui32DataSize);
  pData = malloc(ui32DataSize);
  GetData_Field_M_D (pM, pD, pData, &ui32DataSize);

  //manipulate it as wanted
}
```

4.2.4 Intégration dans un projet continu : CMake

Il est important que les utilisateurs finaux de cette solution puissent la mettre en œuvre de façon simple. Ainsi, nous proposons l'utilisation de CMake qui permet de générer automatiquement les projets pour divers IDE (Integrated Development Environment). Le code Cmake pour utiliser ce prototype afin d'obfusquer automatiquement un protocole nommé "test_running_example" est donnée en annexe C.4.5. L'utilisation de `add_custom_command` permet d'assurer que de nouvelles obfuscations sont bien appliquées à chaque nouvelle compilation. Le prototype prend 4 arguments en paramètre :

- `spec_path` - le chemin vers la spécification du protocole ;
- `nb_obf` - la profondeur du nombre d'obfuscations à appliquer ;
- `seed` - une graine pour initialiser l'algorithme d'aléa, cela est surtout requis afin d'assurer la reproductibilité des expériences effectuées ;

— `log_path` - le chemin vers un fichier de log.

Les fichiers générés contenant le code, le prototype des fonctions ainsi que les structures internes sont automatiquement pris en compte par la partie "*set(generated_files...*". La partie "*set(Test_RunningExample_files...*" définit quant à elle, les fichiers que l'utilisateur doit écrire.

4.3 Différences avec le modèle

Pour faciliter l'implémentation, le prototype présente des différences avec le modèle présenté en chapitre 3. Si le parseur et le sérialiseur proposés dans le chapitre 3 sont génériques et prennent en paramètre le GFM, nous avons pris le parti d'un framework permettant de générer un parseur/sérialiseur spécialisé pour un GFM. Ainsi, ces codes intègrent directement les modifications apportées au GFM par les obfuscations. Le sérialiseur contient les opérations d'obfuscation d'ordonnancement tandis que le parseur intègre les opérations inverses. Les obfuscations d'agrégations sont présentes dans les accesseurs (setters pour les obfuscations, getters pour les opérations de désobfuscations). Lors du parsing, les données sont sauvegardées dans une structure interne permettant l'agrégation finale pour retrouver la donnée d'origine. Lors de la sérialisation, les données produites par l'utilisateur sont sauvegardées dans cette même structure en subissant une désagrégation initiale pour obtenir les données qui seront effectivement sérialisées.

Certaines obfuscations présentent des différences d'implémentation ou sont encore en cours d'implémentation dans le prototype. En effet, les obfuscations `TreeCrypto`, `ReadFromEnd`, et `BoundaryChange` ne sont pas implémentées. Cependant, l'obfuscation `BoundaryChange` a été apliquée manuellement sur l'exemple du protocole `HTTP` qui sert pour les expérimentations du chapitre 5. L'obfuscation `ReadFromEnd` ne présente qu'un changement mineur par rapport à `DissectFromEnd` qui est fonctionnelle. De plus, dans le prototype, les obfuscations `SplitOp`, `SplitCat` et `ConstOp` s'appliquent uniquement à des nœuds terminaux pour simplifier l'implémentation. L'obfuscation `ChildMove` a été implémentée directement de façon macroscopique, en permettant à deux nœuds non accolés d'être échangés, cependant elle ne s'applique que sur des nœuds terminaux.

4.4 Conclusion

Dans ce chapitre, nous avons présenté un exemple d'implémentation de framework qui permet l'application automatique d'obfuscations sur la spécification du protocole. Cette implémentation met en œuvre le modèle d'obfuscations présenté en chapitre 3. De plus, cette solution s'appuie, d'un point de vue architectural, sur des projets de sérialisation de données existants. Ainsi, elle permet à un développeur voulant utiliser une telle solution de sérialisation de données (messages) d'utiliser ce framework. En effet, notre framework est conçu pour que les obfuscations soient appliquées automatiquement et de manière aléatoire sur tous les champs du message,

tandis que l'utilisateur n'a pas à modifier son code, peu importe les obfuscations appliquées. Comme les messages écrits dans les DSL des autres solutions de sérialisation peuvent être écrits dans notre DSL (et inversement), ces travaux sont compatibles avec les projets utilisant de telles solutions. Nos travaux sont évalués dans le chapitre 5.

Chapitre 5

Expérimentations

Sommaire

Ce chapitre est dédié à l'évaluation de l'approche que nous proposons pour l'obfuscation de spécification de protocole. Pour ce faire, nous avons implémenté un prototype présenté en chapitre 4. L'évaluation des performances dépend donc des choix d'implémentations effectués dans ce prototype. Dans un premier temps, nous présentons les protocoles que nous avons choisis pour l'évaluation du prototype. Ensuite, nous proposons un mode opératoire d'évaluation d'un tel framework. Enfin, nous discutons des résultats obtenus.

5.1 Exemples de protocoles

Pour evaluer notre prototype, nous avons choisi deux protocoles qui permettent de couvrir nativement la majorité des cas d'usage du modèle. Ils utilisent notamment : les nœuds terminaux, les séquences de nœuds, les nœuds à taille fixe et les choix avec mots-clés. Le protocole `Tcp-Modbus` est un protocole binaire qui utilise de plus les éléments suivants : répétitions (tableaux), champs disséqués par une taille. Le protocole `HTTP` est un protocole textuel qui utilise en outre les éléments suivants : répétitions (*), champs disséqués par un délimiteur. Nous avons choisi un protocole binaire (`Tcp-Modbus`) et un protocole textuel (`HTTP`) afin d'être représentatifs d'un large panel de protocoles existants. De plus, pour valider le bon fonctionnement du prototype, nous avons créé un protocole binaire dérivé de `TCP-Modbus` dont le format est plus complexe. Il permet de valider que la composition de types de nœuds (par exemple tableaux de champs disséqués par une taille, *i.e.*, tableaux de "Size/Data") fonctionne bien pour la génération du sérialiseur/parseur automatique. Il permet de plus de valider que les obfuscations s'appliquent bien dans le cas de

nœuds composés, notamment que ces obfuscations préservent bien les dépendances `t_path`.

5.1.1 Tcp-Modbus

Nous avons choisi ce protocole car c'est un protocle binaire qui est simple : il possède peu de messages différents, le format de ces messages comporte peu de champs différents. Même s'il n'y a pas d'obfuscations de la grammaire du protocole, nous avons pris en considération le fait que la grammaire de ce protocole est de type question réponse, et par conséquent simple à implémenter. La spécification complète de ce protocole dans le DSL est donné en annexe B.1. Nous avons choisi 16 messages de la spécification correspondant à l'implémentation du client *simply modbus*[1]. Cependant, le protocole `Tcp-Modbus` est particulier en cela qu'il garde le même identifiant de messages pour chaque message envoyé au serveur et sa réponse associée. Le message du client et la réponse du serveur contiennent en fait la même valeur pour le `Leaf:key` *Fc*. Cela se retrouve dans l'extrait ci-dessous (liste 5.1) où nous avons 2 types de messages (mot-clé 01 et 02), mais la question (MFc) et la réponse (RFc) partagent le même mot-clé. En pratique, nous avons considéré que les questions constituaient un protocole et les réponses un autre protocole. La concordance entre les questions et les réponses fait partie de la grammaire et n'est pas traitée dans cette thèse.

Listing 5.1 – Modèle pour le protocole TCP-Modbus (4 messages) exprimé dans notre DSL

```
Mfc01 := Fc Ref Coil
Fc := [Fixed,1] : E : Keyword,01
Ref := [Fixed,2] : E : Data
Coil := [Fixed,2] : E : Data

Rfc01 := Fc Count Coils
Fc := [Fixed,1] : E : Keyword,01
Count := [Fixed,1] : E : Counter
Coils := [Child] : T : Coil{Count}
Coil := [Fixed,1] : E : Data

Mfc02 := Fc Ref Inputs
Fc := [Fixed,1] : E : Keyword,02
Ref := [Fixed,2] : E : Data
Inputs := [Fixed,2] : E : Data

Rfc02 := Fc Count Inputs
Fc := [Fixed,1] : E : Keyword,02
Count := [Fixed,1] : E : Counter
```

1. http ://www.simplymodbus.ca

```
Inputs := [Child] : T : Input{Count}
Input := [Fixed,1] : E : Data

...
```

5.1.2 HTTP

Le protocole HTTP a été choisi car il est largement utilisé dans la littérature pour évaluer les outils de rétro-conception de protocole. Nous avons implémenté un seul type de messages générique d'après sa norme [Fielding 2014] et qui permet de représenter l'ensemble des messages HTTP. Le détail est donné en liste 5.2. Cependant, l'utilisation de délimiteurs limite grandement l'application d'obfuscations. Ainsi, nous avons appliqué dans un premier temps une première passe de la transformation BoundaryChange sur tous les champs du message, le résultat est exprimé dans le DSL présenté en liste 5.3.

Listing 5.2 – Modèle pour le protocole HTTP exprimé dans notre DSL

```
HttpRequest := RequestLine rHeaders Body
RequestLine := [Delimiter, CRLF] : L : Method URI ↩
   Version
rHeaders := [Delimiter, CRLFCRLF] : R : Header*
Header := [Delimiter, CRLF] : L : Key Value
Key := [Delimiter, SP] : E : Data
Value := [Delimiter, Parent] : E : Data
Body := [Delimiter, Parent] : E : Data
Method := [Delimiter, SP] : E : Data
URI := [Delimiter, SP] : E : Data
Version := [Delimiter, Parent] : E : Data
```

Listing 5.3 – Modèle pour le protocole HTTP une fois les délimiteurs enlevés exprimé dans notre DSL

```
HttpRequest := RequestLine Count rHeaders BodySz Body
RequestLine := [Child] : L : MethodSz Method URISz URI ↩
   VersionSz Version
Count := [Fixed, 2] : E : Counter
rHeaders := [Child] : T : Header{Count}
Header := [Child] : L : KeySz Key ValueSz Value
KeySz := [Fixed, 2] : E : Length
ValueSz := [Fixed, 2] : E : Length
Key := [Length, {KeySz}] : E : Data
Value := [Length, {ValueSz}] : E : Data
MethodSz := [Fixed, 2] : E : Length
URISz := [Fixed, 2] : E : Length
```

```
VersionSz := [Fixed, 2] : E : Length
Method := [Length, {MethodSz}] : E : Data
URI := [Length, {URISz}] : E : Data
Version := [Length, {VersionSz}] : E : Data
BodySz := [Fixed, 2] : E : Data
Body := [Length, {BodySz}] : E : Data
```

5.1.3 Running test

Ce protocole, dérivé de `Tcp-Modbus`, a été créé afin de tester des cas plus complexes. Pour l'évaluation du bon fonctionnement du prototype, nous avons modifié le message Fc_{16} pour valider la composition d'un tableau (`Gfm_repetition`) avec des `t_path` (en pratique, un tableau de Size/Data) plutôt que d'avoir simplement un tableau de données à taille fixe. Ce message est présenté en liste 5.4. De plus, nous avons créé d'autres messages plus simples afin de valider de manière la plus unitaire possible le bon fonctionnement du prototype. Enfin, nous avons aussi créé des messages complexes mettant en œuvre la composition des types et des obfuscations. Ainsi, nous avons aussi testé les tableaux de tableaux (`Gfm_repetition` contenant un `Gfm_repetition`) et les listes de tableaux (`Gfm_star` contenant un `Gfm_repetition`). Une composition plus profonde n'a pas été évaluée, car les problématiques de connaissance du chemin (donnée avant obfuscation) et des références (`t_path`) jusqu'au nœud de traitement apparaissent dès les premières compositions.

Listing 5.4 – Modèle pour le protocole `Running test` dans notre DSL

```
M := Fc Count Coils
Fc := [Fixed,1] : E : Keyword,10
Count := [Fixed,1] : E : Counter
Coils := [Child] : T : Coil{Count}
Coil := [Child] : L : S D
S := [Fixed,1] : E : Length
D := [Length,{S}] : E : Data

...
```

5.2 Évaluation du fonctionnement du prototype

Dans un premier temps, nous avons cherché à valider le bon fonctionnement du prototype. Pour cela, nous avons défini un certain nombre de messages de tests, dans le protocole `Running test`, afin de couvrir d'une part les différents types de nœuds de manière unitaire, et d'autre part une large variété de compositions de types de nœuds. De plus, pour chaque type de nœud, nous avons appliqué diverses obfuscations, puis nous avons vérifié qu'elles étaient bien appliquées sur le GFM. Ensuite, nous avons évalué que le code généré produisait le même résultat du point de

vue d'un utilisateur de cette solution et que les données sérialisées correspondaient bien à l'obfuscation appliquée. Cependant, cette dernière étape était manuelle et ne couvre actuellement pas l'ensemble des obfuscations pour l'ensemble des types de nœuds. À ce jour, les tests unitaires et d'intégration sont bien développés pour un prototype mais nécessitent plus de travail pour être industrialisés. Ainsi, nous avons pu valider le bon fonctionnement d'une grande partie du prototype.

La liste 5.5 montre le résultat du message donné en liste 5.4 avec une obfuscation par nœud. Le DSL n'est pas prévu pour afficher les obfuscations que nous pratiquons, ainsi nous avons ajouté un champ obfuscation pour cet exemple. Si l'obfuscation est `None`, alors il n'y a pas d'autres obfuscations appliquées. Il est à noter que dans cet exemple, le dernier nœud reçoit l'obfuscation `DissectFromEnd`. Cette obfuscation aura pour incidence d'inverser l'ordre de lecture des différents D du tableaux. En effet, s'il y a 3 élements dans le tableau, alors la sérialisation (des S et D) serait : $S\_3A1D75A0_1, S\_3A1D75A0_2, S\_3A1D75A0_3, D_3, D_2, D_1$ avec la taille de D_i qui est donné par $S\_3A1D75A0_i + 0x37$.

Listing 5.5 – Modèle pour le protocole `Running test` dans notre DSL avec une obfuscation par nœud appliquée

```
M := Fc Count Coils
Fc := [Fixed,1] : E : ConstAdd(Fc_8808D707,0x28) : ↩
   Keyword,10
Fc_8808D707 := [Fixed,1] : E : None : Data
Count := [Fixed,1] : E : SplitXor(Count_F8122599,↩
   Count_AEEBDD80): Counter
Count_F8122599 := [Fixed,1] : E : None : Data
Count_AEEBDD80 := [Fixed,1] : E : None : Data
Coils := [Child] : T : RepSplit(Coils_EE64A6E4,↩
   Coils_1925892) : {Count}
Coils_EE64A6E4 := [Child] : T : None : S{Count}
S := [Fixed,1] : E : ConstSub(S_3A1D75A0,0x37) : Length
S_3A1D75A0 := [Fixed,1] : E : None : Data
Coils_1925892 := [Child] : T : None : D{Count}
D := [Length,{S}] : E : DissectFromEnd : Data
```

5.3 Évaluation des performances du prototype

Dans un second temps, nous avons cherché à évaluer les performances du prototype, et plus particulièrement, le coût qu'engendrent de telles obfuscations par rapport aux bénéfices obtenus. Il est difficile d'évaluer la qualité d'une obfuscation face à la rétro-conception. Nous pouvons néanmoins utiliser les critères de [Collberg 1997] à l'obfuscation de spécification de protocole :

— *Puissance* - à quel point le programme (ici le protocole) est plus difficile à comprendre pour un humain ;

— *Résistance* - à quel point le programme (ici le protocole) est plus difficile à analyser pour des outils automatiques ;
— *Coût* - la surcharge en temps d'exécution ou de mémoire utilisée engendrée par l'obfuscation ;

Nous pouvons mesurer la *puissance* des obfuscations via l'augmentation de la complexité du code du parseur et du sérialiseur associés. C'est un bon indicateur car elle reflète effectivement la complexité de la structure du message. De plus, nous cherchons à rendre la rétro-conception plus difficile, ce qui est la cas pour une complexité de code accrue. Cependant, un analyste en rétro-conception aura généralement seulement accès au code compilé et non au code source. Ainsi, il faut aussi choisir des critères de complexité de code qui se reflètent dans le code compilé. Pour ce qui est de la *résistance* des obfuscations, très peu d'outils de rétro-conception de protocole sont disponibles et fonctionnels. Nous avons donc aussi laissé ce critère pour des travaux futurs. Le *coût* peut être évalué suivant quatre critères : la difficulté d'utilisation d'une telle solution ; le temps de génération du code de parseur/sérialiseur ; le temps d'exécution du code de parseur/sérialiseur ; mais surtout, pour le cas de protocole réseaux, l'augmentation de la taille du message sérialisé car nous ne voulons pas engorger les réseaux de communication.

5.3.1 Mode opératoire

Nous ne cherchons pas à mesurer l'impact de chaque obfuscation prise unitairement. En effet, une obfuscation possède souvent un impact faible sur le protocole, par exemple *SplitCat* produit le même résultat sérialisé même si le GFM change. C'est bien la composition des obfuscations qui permet une plus forte protection. Ainsi, les différentes mesures que nous effectuons se font par rapport à une mesure de référence non obfusquée. Ensuite, nous effectuons des mesures en fonction du degré de composition des obfuscations. Nous appliquons une obfuscation aléatoire sur chaque nœud du GFM, obtenons un GFM obfusqué, puis nous répétons cette opération sur le GFM obfusqué. Chaque répétition permet d'obtenir une profondeur supplémentaire dans la composition des obfuscations. Ainsi, nous avons de 0 à 4 degrés d'obfuscations. Pour mesurer les différents résultats, nous avons choisi à chaque fois 10 graines pour appliquer les différentes obfuscations, ainsi que 100 messages générés de manière aléatoire, pour chaque protocole (`Tcp-Modbus` et `HTTP`). Cela correspond à 5000 expérimentations pour chaque protocole.

La première mesure est la taille du GFM obfusqué. En effet, comme les obfuscations sont générées aléatoirement, le nombre d'obfuscations appliquées peut légèrement varier d'une expérimentation à une autre. Nous mesurons la *puissance* des obfuscations par la complexité du code généré par le prototype. Pour les mesures de complexité de code, nous mesurons le nombre de lignes de code générées, le nombre de structures internes utilisées, la taille et la profondeur du graphe d'appels ainsi que le nombre d'allocations réalisées lors du parsing d'un message. Ce dernier paramètre peut également être considéré dans l'évaluation du coût, mais rappelons que l'on cherche à complexifier la tâche de rétro-conception. Dans cette optique,

un grand nombre d'allocations mémoire complexifie l'analyse du code ou le suivi du flot de données. Cependant, il a bien également un impact important sur le temps d'exécution. Dans notre implémentation, nous avons choisi d'allouer toutes nos structures dynamiquement. Cela présente l'avantage d'une grande souplesse de code mais implique un grand nombre d'allocations/libérations de mémoire, d'où l'impact négatif sur le temps d'exécution. Nous avons instrumenté un allocateur pour d'une part mesurer le nombre d'allocations effectuées et d'autre part nous assurer qu'il n'y avait pas de fuites mémoires. Ce dernier point n'est malheureusement pas valide et devra être corrigé lors de l'industrialisation. Les graphes d'appels sont obtenus grâce à l'outil *cflow* appliqué sur la fonction de parsing. Les *coûts* sont évalués lors de la génération du code ou de son exécution.

5.3.2 Benchmark

Les résultats sont résumés pour chaque protocole dans les tableaux 5.1 et 5.2, respectivement. Trois valeurs sont indiquées avec la syntaxe suivante : *moyenne, [min, max]*. Les résultats reportés pour les mesures de puissance sont normalisés par les valeurs associées avec la version non obfusquée. Les mesures de coûts sont données en valeur absolue.

Pour le cas simple où au plus une obfuscation par nœud est appliquée (ce qui correspond à une moyenne de 10.1 obfuscations appliquées pour `HTTP` et de 47.8 obfuscations appliquées pour `Tcp-Modbus`), la complexité du code généré est environ deux fois celle du code sans obfuscations. En particulier, l'augmentation du nombre de structures reflète une différence significative entre la spécification d'origine et le résultat obfusqué. Pour les autres mesures, elles augmentent conformément à ce qui était attendu. Seul le nombre d'allocations subit une franche augmentation, qui correspond au choix d'implémentation utilisant de nombreuses structures dynamiques différentes.

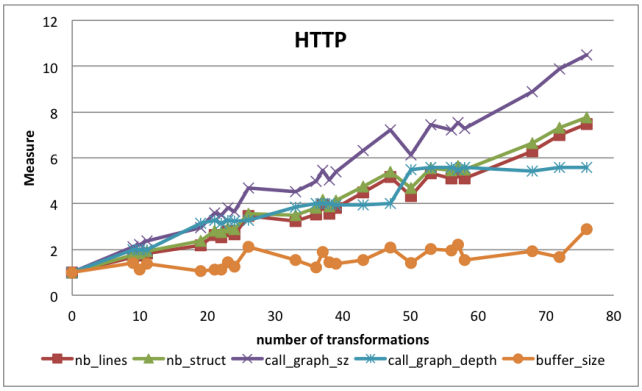

FIGURE 5.1 – HTTP : mesures de *puissance* normalisées

Les figures 5.1 et 5.2 montrent l'évolution relative (à la mesure non-obfusquée) des mesures de *puissance* en fonction du nombre d'obfuscations appliquées. Nous observons une augmentation linéaire du nombre de lignes de code, du nombre de

Nb. transf. par nœud	1	2	3	4
Nb. transf. appliquées	47[45; 51]	107[101; 112]	184[167; 200]	279[258; 310]
Puissance (normalisée)				
Nb. lignes	1.9[1.8; 2.0]	3.0[2.8; 3.2]	4.5[4.1; 4.9]	6.4[5.7; 7.1]
Nb. struct.	1.9[1.8; 1.9]	2.9[2.7; 3.1]	4.3[3.9; 4.7]	6.0[5.4; 6.6]
Graphe d'appel (taille)	2.6[2.1; 3.2]	4.3[3.4; 5.5]	6.8[4.7; 8.6]	9.8[6.8; 12.2]
Graphe d'appel (prof.)	2.0[2.0; 2.0]	2.5[2.5; 2.5]	3.3[3.3; 3.3]	3.8[3.8; 3.8]
Nb allocations	4.71[0.81; 23.13]	11.33[1.81; 69.04]	23.27[3.80; 114.49]	40.46[5.79; 195.52]
Coûts (absolus)				
Génération (ms)	6.39[5.97; 6.72]	12.53[9.66; 31.06]	16.34[14.56; 17.74]	24.29[21.76; 27.01]
Parsing (ms)	0.01[0.00; 0.06]	0.03[0.01; 0.14]	0.05[0.01; 0.25]	0.09[0.02; 0.52]
Sérialisation (ms)	0.01[0.00; 0.06]	0.02[0.00; 0.10]	0.03[0.01; 0.16]	0.05[0.01; 0.31]
Taille du buffer (octets)	30[3; 195]	33[3; 293]	38[3; 381]	42[3; 478]

TABLE 5.1 – Résultats comparatifs pour le protocole `TCP-Modbus`

structures utilisées ainsi que de la taille du graphe d'appels. L'augmentation de la profondeur du graphe d'appels ainsi que de la taille du buffer est plus douce et tend à se stabiliser. Plus particulièrement, l'augmentation de la taille du buffer sérialisé

Nb. transf. par nœud	1	2	3	4
Nb. transf. appliquées	10[9; 11]	22[19; 26]	39[33; 47]	59[50; 76]
Puissance (normalisée)				
Nb. lignes	1.7[1.6; 2.0]	2.7[2.2; 3.5]	4.0[3.2; 5.2]	5.6[4.3; 7.5]
Nb. struct.	1.8[1.7; 2.1]	2.9[2.4; 3.6]	4.3[3.5; 5.4]	5.9[4.7; 7.8]
Graphe d'appel (taille)	2.2[2.0; 2.6]	3.7[3.0; 4.7]	5.6[4.5; 7.2]	7.9[6.1; 10.5]
Graphe d'appel (prof.)	2.0[2.0; 2.0]	3.2[3.1; 3.3]	4.0[3.9; 4.0]	5.5[5.4; 5.6]
Nb allocations	9.21[6.91; 13.70]	32.07[16.89; 58.73]	83.74[52.16; 158.80]	193.07[118.45; 397.80]
Coûts (absolus)				
Génération (ms)	2.10[1.92; 2.41]	3.17[2.59; 4.03]	4.80[3.84; 6.36]	8.93[5.41; 26.08]
Parsing (ms)	0.06[0.04; 0.12]	0.15[0.08; 0.47]	0.37[0.22; 1.00]	0.79[0.47; 2.80]
Sérialisation (ms)	0.04[0.02; 0.10]	0.10[0.05; 0.34]	0.22[0.13; 0.75]	0.43[0.25; 1.57]
Taille du buffer (octets)	137[95; 244]	154[101; 284]	181[112; 297]	219[119; 404]

TABLE 5.2 – Résultats comparatifs pour le protocole `HTTP`

reste faible ce qui est très important dans notre cas où le traffic réseau est une

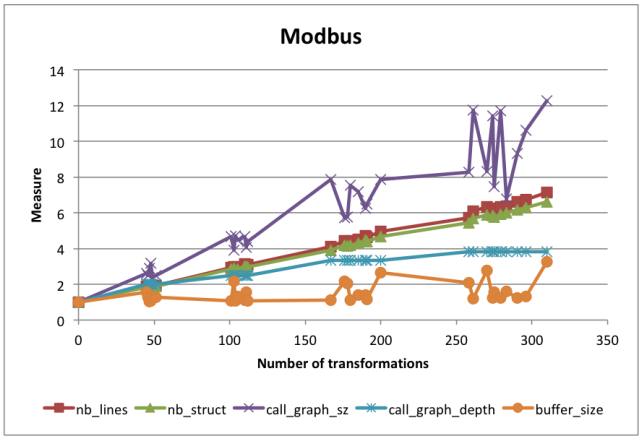

FIGURE 5.2 – Modbus : mesures de *puissance* normalisées

ressource beaucoup plus contrainte que le temps d'exécution.

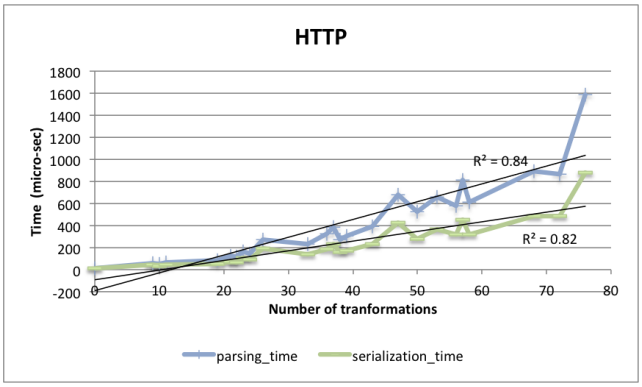

FIGURE 5.3 – HTTP : temps de parsing et de serialisation

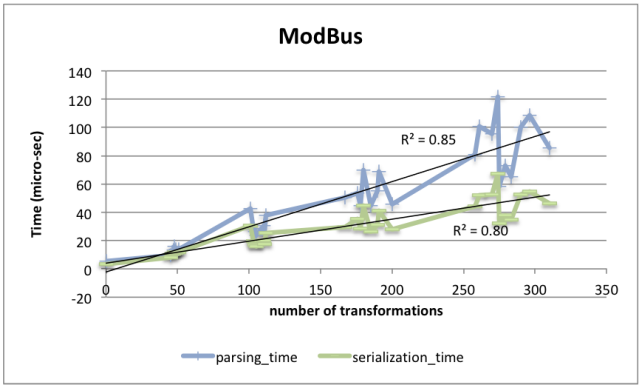

FIGURE 5.4 – Modbus : temps de parsing et de serialisation

Le coût des obfuscations est illustré en figures 5.3 et 5.4 qui présentent l'évolution du temps d'exécution en fonction du nombre d'obfuscations appliquées sur le

GFM. La ligne noire indique le résultat de la régression linéaire entre ces temps et le nombre d'obfuscations (le coefficient de corrélation est aussi indiqué). Ces figures montrent que le temps d'exécution augmente effectivement avec le nombre d'obfuscations. Cependant, cette augmentation est principalement due à notre choix d'implémentation à base de structures dynamiques. De plus, cette augmentation reste mesurée. Il est à noter que ces résultats sont obtenus pour un grand nombre d'obfuscations. Un développeur pourrait considérer suffisant un nombre restreint d'obfuscations. Les temps d'exécution restent cohérents avec une application de parsing : inférieur à 0.5ms pour `Tcp-Modbus` et inférieur à 2.8ms pour `HTTP`. De plus, les valeurs moyennes sont significativement plus faibles.

Enfin, le coût de génération du projet associé à ce prototype reste bas. Le temps maximal de génération du projet reste inférieur à 4ms dans le pire des cas. Ce pire cas correspond à une succession de *SplitOp* appliquées sur un champ de taille conséquente. Ce temps de génération est moins critique que celui d'exécution.

5.4 Conclusion

Dans ce chapitre, nous avons présenté une évaluation de l'obfuscation de la spécification de protocole basée sur un prototype présenté en chapitre 4. Comme nous appliquons des obfuscations de protocole dans le paradigme de l'obfuscation logicielle, nous utilisons les mêmes mesures types utilisées pour l'évaluation des obfuscations logicielles. Ainsi, les mesures de complexité de code appliquées sur le code généré reflètent la qualité des obfuscations. Cependant, les obfuscations que nous proposons, prises indépendamment les unes des autres, ont des effets limités. C'est bien la composition de ces obfuscations qui permet une bonne efficacité de l'obfuscation. C'est pourquoi, nous avons évalué l'impact de cette composition d'obfuscation. Les résultats montrent une augmentation significative de la complexité du code avec le degré de composition d'obfuscations, tandis que l'augmentation du coût de l'utilisation reste acceptable. Ainsi, cette approche est légitime et permet d'apporter une nouvelle classe d'obfuscations de protocoles qui s'applique directement sur sa spécification.

Ce sont des travaux initiaux qui méritent d'être complétés et développés pour aborder les nouveaux problèmes ouverts par ces travaux. En effet, nous avons demandé à un expert (et contributeur) de *Netzob* [Bossert 2014a, Bossert 2014b] (un outil de rétro-conception de protocoles basé sur l'analyse de traces réseaux) de pratiquer la rétro-conception de protocole sur un protocole obfusqué par notre approche. La trace réseaux contenait 4 types de messages différents du protocol `TCP-Modbus` et leurs réponses associées. En moins d'une demi-heure, il a été capable de retrouver le format de ces messages non obfusqués. Pour la trace du protocole obfusqué (contenant les mêmes données échangées), avec seulement une obfuscation par nœud, il n'a pas obtenu de résultats concluant après 2 heures de travail. Il a confirmé que ce protocole obfusqué était plus difficile à analyser avec les outils classique de rétro-conception de protocole. Bien évidemment, cette constation n'est pas suffisante

et nécessite une évaluation approfondie. Cependant, une telle évaluation n'est pas simple car elle requiert la contribution d'experts indépendant en rétro-conception de protocoles, ainsi qu'un accès à divers outils de rétro-conception de protocle. Comme nous l'avons fait remarquer en chapitre 1, très peu d'outils sont disponibles.

Conclusion

De nombreux protocoles applicatifs de communication différents sont utilisés par les systèmes. Pour certains d'entre eux, leur spécification est publique et disponible pour faciliter entre autres l'interopérabilité des systèmes. Pour les autres, la spécification est gardée secrète pour différentes raisons, allant de la protection de la propriété intellectuelle jusqu'à l'intention de limiter les composants qui peuvent participer aux échanges. Toutefois, ce choix de protection par l'obscurité doit se confronter aux méthodes et outils de rétro-conception existantes qui sont de plus en plus nombreux. Ainsi, à partir d'une observation du comportement des composants qui communiquent et des échanges entre ces composants, des outils de rétro-conception arrivent à déduire un modèle du protocole de communication proche de la spécification de départ. Dans ce contexte, des approches d'obfuscation des échanges peuvent être mises en place pour augmenter la difficulté de rétro-conception. Les approches existantes reposent sur des techniques de masquage de l'information qui conduisent à des défis régulièrement surmontés par les outils de rétro-conception. Dans les travaux présentés dans cette thèse, nous proposons une méthode d'obfuscation qui s'applique directement sur la spécification dans l'objectif d'augmenter la difficulté de rétro-conception en changeant la classe de complexité du problème. Nous avons basé notre approche sur les observations suivantes :

— les algorithmes de rétro-conception de protocoles s'appuient sur une étape préliminaire de classification des messages. Ainsi, une obfuscation qui aboutit à une mauvaise classification affecte négativement la qualité de la rétro-conception du protocole ;

— la spécification des protocoles est généralement basée sur des modèles réguliers (automates, arbres, ...) qui sont simples à implémenter et pour lesquels des algorithmes efficaces de parsing et de sérialisation existent. Naturellement, les outils de rétro-conception de protocoles adoptent des modèles similaires pour l'inférence de protocoles. Ces outils sont inefficaces si des modèles plus complexes (automates à pile, ...) sont utilisés sans sacrifier le temps d'exécution des parser et serializer ;

— les solutions d'obfuscations de protocoles existantes s'appliquent après la phase de développement. Cette façon de faire introduit naturellement une interface contenant le message en clair avant son obfuscation. Une sonde (un débogueur par exemple) peut être placée au niveau de cette interface pour obtenir le message en clair et effectuer une rétro-conception classique. L'intégration des techniques d'obfuscation durant le développement permet d'obfusquer à la volée les messages pour empêcher qu'ils soient stockés entièrement en mémoire ;

— le développement, le débogue et la maintenance de protocoles obfusqués ne doivent pas constituer, pour l'utilisateur d'une telle solution, un surcoût important. Ainsi, la construction des messages obfusqués doit utiliser les

mêmes API que pour le message en clair.

L'approche et le prototype réalisé respectent l'ensemble de ces observations. Nous avons étudié les différents outils de rétro-conception de protocole associés à plus de 32 publications parues durant ces 14 dernières années. Dans un premier temps, nous avons discuté les défis qui sont liés à la rétro-conception de protocole. Cela permet de proposer des obfuscations qui s'appuient spécifiquement sur ces défis pour rendre la rétro-conception plus difficile. De plus, nous avons proposé une nouvelle classification des outils, basée non-seulement sur le type d'analyse effectuée (une analyse des traces réseaux ou d'une application implémentant le protocole), mais aussi sur le type d'inférence actif ou passif. Cette étude est la première contribution de cette thèse.

Dans un second temps, nous avons proposé une nouvelle approche des obfuscations de protocoles. La clé de ces obfuscations est de s'appuyer sur les défis de la rétro-conception, tels que l'inférence du modèle, la délimitation des champs ou le clustering des messages pour s'assurer que le modèle inféré soit le plus erroné possible. Nous proposons un nouveau modèle que l'on nomme *Graphe de Format de Message* (GFM) pour le format des messages qui permet explicitement l'application des obfuscations. Plusieurs types d'obfuscations atomiques sont proposées. Ces obfuscations peuvent être composées de différentes manières et appliquées aux différents nœuds du GFM. Elles ont été conçues pour être inversibles. Le fait d'avoir des obfuscations atomiques permet de simplifier grandement la tâche de vérification formelle ainsi que l'implémentation d'un prototype. Nous avons effectué la vérification formelle sur certaines obfuscations, les obfuscations restantes devraient pouvoir être vérifiées de manière identique. Ce modèle et les obfuscations associées constituent la seconde contribution de cette thèse.

Dans le même temps, nous avons proposé un nouveau concept où la spécification initiale du protocole est donnée par un utilisateur de notre solution. Comme pour les outils de sérialisation existants (ProtoBuff, Cap'n'Proto, thrift...), le code pour le parseur et pour le sérialiseur est généré automatiquement pour cette spécification. Des obfuscations différentes sont appliquées à chaque compilation pour obtenir un nouveau protocole qui est plus complexe à inférer que le protocole d'origine. L'originalité de notre solution est de garder les mêmes interfaces de parsing et de sérialisation tout en assurant que le code du parseur et du sérialiseur contient les obfuscations associées au nouveau protocole. Ce concept est validé par un prototype qui sert aussi pour l'évaluation pratique des performances des obfuscations. Cette approche peut aussi être utilisée pour les projets basés sur les outils de sérialisation cités ci-dessus, soit en implémentant les obfuscations dans ces projets, soit en ajoutant les glues logicielles pour leur DSL et leurs interfaces de construction des messages. Ce prototype est la troisième contribution de cette thèse.

L'évaluation des obfuscations a été effectuée à partir des mesures utilisées habituellement en obfuscation logicielle. Les résultats montrent une nette augmentation de la complexité du parseur et du sérialiseur et par conséquent du protocole associé tandis que le coût reste borné pour ce qui est de la taille du buffer sérialisé. Seul le nombre d'allocations effectuées durant le parsing d'un message subit une forte

augmentation, mais cela est dû au choix de l'implémentation à base de multiples structures dynamiques. Ce résultat est par ailleurs cohérent avec notre objectif de rendre la rétro-conception plus complexe. Le pendant à ce choix d'implémentation est une augmentation du temps d'exécution du parseur et du sérialiseur. Cette évaluation est la dernière contribution de cette thèse.

Perspectives

Avec cette nouvelle classe d'obfuscations de protocoles, nous commençons à couvrir l'obfuscation de protocoles au niveau de sa spécification. Cependant, il y a encore de nombreux travaux à effectuer dans ce domaine. En effet, la validation formelle des obfuscations est en cours et doit être terminée. De même, des améliorations du prototype actuel sont aussi envisagées (tests systématiques, modification de l'architecture pour laisser à l'utilisateur le choix de l'algorithme d'application des obfuscations unitaires). Notre objectif est de diffuser plus largement en service libre.

De plus, nous nous sommes appliqués à évaluer l'impact de la composition des obfuscations plutôt que l'impact de chaque obfuscation unitairement. Ainsi, nous avons l'intuition que l'algorithme de choix des obfuscations actuel (un choix aléatoire), s'il a le mérite de créer un nouveau protocole obfusqué à chaque compilation, devrait d'être amélioré. En effet, actuellement, il ne contient aucune intelligence alors que certaines obfuscations sont plus efficaces sur certains champs. Par exemple, l'obfuscation `SplitOp` transforme un champ n en deux champs $n1$ et $n2$ dont le contenu d'origine est obtenu par une opération mathématique sur le contenu des deux champs créés, i.e. pour une addition, $n = n1 + n2$. L'application de cette obfuscation sur des champs *mots-clé* (`Leaf:key`) permet de modifier le clustering en enlevant les éléments fixes du message. Ainsi un compromis entre la diversification des protocole (choix différent des obfuscations à chaque compilation) et l'efficacité des obfuscations est à étudier.

Nous avons envoyé un extrait de protocole obfusqué à l'un des auteurs de l'outil Netzob, et il a confirmé que le protocole obfusqué est plus difficile à rétro-concevoir. Cependant, cette observation n'est pas suffisante. Ainsi, une évaluation plus extensive basée sur des experts indépendants, ainsi que sur une large gamme d'outils de rétro-conception de protocoles est envisagée. De plus, une plus grande variété de protocoles pourrait être envisagée.

Dans l'absolue, notre approche n'empêche pas l'attaquant de rétro-concevoir le protocole de communication s'il y consacre le temps nécessaire, même si notre approche rend inefficace les outils automatiques et si elle oblige l'attaquant à consacrer un temps qui devient considérable. Par conséquent, une extension que nous envisageons vise à mettre à jour régulière les obfuscations réalisées sur le protocole de communication. Bien entendu, la période entre les mises à jour doit être plus courte que le temps nécessaire à la rétro-conception. Un défis pour cette extension concerne la mise à jour à chaud, sans interruption de service, pour les installations critiques.

Enfin, cette thèse traite de l'obfuscation de format de message uniquement. Il nous semble intéressant d'étendre notre approche à l'obfuscation de la grammaire de protocole en suivant le même principe. Souvent, la grammaire est représentée par un automate. Effectivement, dans une trace réseau, la compréhension des différents messages qui la composent, repose exclusivement sur la compréhension des messages précédents. Si une obfuscation permet d'émettre un message ou une partie d'un message en avance dans les échanges (connaissant les motivations du programme), alors la compréhension de ce morceau de message doit forcément se baser également sur les messages futurs. Nous avons l'intuition que cette approche peut également rendre plus difficile la rétro-conception de la grammaire du protocole.

Annexe A

Parse et serialize

A.1 Parse

```
let rec parse_star f = function
| Nil -> Some Nil
| Cons (b, l) ->
  (match f (Cons (b, l)) with
   | Some p ->
     let Pair (p0, rem) = p in
     let Pair (p1, _) = p0 in
     let Pair (ast, _) = p1 in
     (match lt_dec (length rem) (length (Cons (b, l))) with
      | Left ->
        (match parse_star f rem with
         | Some asts -> Some (Cons (ast, asts))
         | None -> None)
      | Right -> None)
   | None -> None)

let rec parse_repetition f num msg =
  match num with
  | O -> Some (Pair ((Pair (Nil, Nil)), msg))
  | S num0 ->
    (match f msg with
     | Some p ->
       let Pair (p0, rem) = p in
       let Pair (p1, read) = p0 in
       let Pair (ast, _) = p1 in
       (match parse_repetition f num0 rem with
        | Some p2 ->
          let Pair (p3, rem0) = p2 in
          let Pair (asts, reads) = p3 in
          Some (Pair ((Pair ((Cons (ast, asts)),
                  (app read reads))), rem0))
        | None -> None)
     | None -> None)
```

```
let rec parse mem gfm msg =
  match gfm with
  | Gfm_leaf ->
    let size = bytes_to_nat mem in
    (match headn size msg with
     | Some p ->
       let Pair (a, rem) = p in
       Some (Pair ((Pair ((Pair ((Ast_leaf a), a)), a)), rem))
     | None -> None)
  | Gfm_leaf_fixed size ->
    (match headn size msg with
     | Some p ->
       let Pair (a, rem) = p in
       Some (Pair ((Pair ((Pair ((Ast_leaf a), a)), a)), rem))
     | None -> None)
  | Gfm_dis gfm0 ->
    let size = bytes_to_nat mem in
    (match headn size msg with
     | Some p ->
       let Pair (msg0, rem) = p in
       (match parse mem gfm0 msg0 with
        | Some p0 ->
          let Pair (p1, b) = p0 in
          (match b with
           | Nil -> Some (Pair (p1, rem))
           | Cons (_, _) -> None)
        | None -> None)
     | None -> None)
  | Gfm_dis_fixed (size, gfm0) ->
    (match headn size msg with
     | Some p ->
       let Pair (msg0, rem) = p in
       (match parse mem gfm0 msg0 with
        | Some p0 ->
          let Pair (p1, b) = p0 in
          (match b with
           | Nil -> Some (Pair (p1, rem))
           | Cons (_, _) -> None)
        | None -> None)
     | None -> None)
  | Gfm_delim gfm0 ->
    (match substring byte_eq_dec mem msg with
     | Some p ->
       let Pair (prefix0, suffix) = p in
```

```
      (match parse Nil gfm0 prefix0 with
       | Some p0 ->
         let Pair (p1, b) = p0 in
         let Pair (p2, _) = p1 in
         (match b with
          | Nil -> Some (Pair ((Pair (p2, (app prefix0 mem))),
                               suffix))
          | Cons (_, _) -> None)
       | None -> None)
   | None -> None)
| Gfm_sequence (g1, g2) ->
  (match parse mem g1 msg with
   | Some p ->
     let Pair (p0, rem) = p in
     let Pair (p1, r1) = p0 in
     let Pair (a1, m1) = p1 in
     (match parse m1 g2 rem with
      | Some p2 ->
        let Pair (p3, rem0) = p2 in
        let Pair (p4, r2) = p3 in
        let Pair (a2, m2) = p4 in
        Some (Pair ((Pair ((Pair ((Ast_sequence (a1, a2)), m2)),
        (app r1 r2))), rem0))
      | None -> None)
   | None -> None)
| Gfm_repetition gfm0 ->
  let num = bytes_to_nat mem in
  (match parse_repetition (parse Nil gfm0) num msg with
   | Some p ->
     let Pair (p0, rem) = p in
     let Pair (asts, reads) = p0 in
     Some (Pair ((Pair ((Pair ((Ast_repetition asts), Nil)),
             reads)), rem))
   | None -> None)
| Gfm_star gfm0 ->
  let size = bytes_to_nat mem in
  (match headn size msg with
   | Some p ->
     let Pair (msg0, rem) = p in
     (match parse_star (parse Nil gfm0) msg0 with
      | Some asts ->
        Some (Pair ((Pair ((Pair ((Ast_star asts), Nil)),
                msg0)), rem))
      | None -> None)
```

```
    | None -> None)
  | Gfm_choice (v, g1, g2) ->
    let w = bytes_to_nat mem in
    (match Nat.eq_dec v w with
     | Left ->
       (match parse mem g1 msg with
        | Some p ->
          let Pair (p0, rem) = p in
          let Pair (p1, read) = p0 in
          let Pair (ast, mem0) = p1 in
          Some (Pair ((Pair ((Pair ((Ast_choice ast), mem0)),
                  read)), rem))
        | None -> None)
     | Right ->
       (match parse mem g2 msg with
        | Some p ->
          let Pair (p0, rem) = p in
          let Pair (p1, read) = p0 in
          let Pair (ast, mem0) = p1 in
          Some (Pair ((Pair ((Pair ((Ast_choice ast), mem0)),
                  read)), rem))
        | None -> None))
  | Gfm_fct fct0 -> fct0.fct_inv mem msg
```

A.2 Serialize

```
let rec serialize_repetition f = function
| Nil -> Some Nil
| Cons (ast, asts0) ->
  (match f ast with
   | Some p ->
     let Pair (_, msg) = p in
     (match serialize_repetition f asts0 with
      | Some msgs -> Some (app msg msgs)
      | None -> None)
   | None -> None)

let rec serialize_star f = function
| Nil -> Some Nil
| Cons (ast, asts0) ->
  (match f ast with
   | Some p ->
     let Pair (_, msg) = p in
     (match Nat.eq_dec (length msg) O with
```

```
      | Left -> None
      | Right ->
        (match serialize_star f asts0 with
         | Some msgs -> Some (app msg msgs)
         | None -> None))
  | None -> None)

let rec serialize mem gfm ast = match ast with
| Ast_leaf a ->
  (match gfm with
   | Gfm_leaf ->
     let size = bytes_to_nat mem in
     (match Nat.eq_dec size (length a) with
      | Left -> Some (Pair (a, a))
      | Right -> None)
   | Gfm_leaf_fixed size ->
     (match Nat.eq_dec size (length a) with
      | Left -> Some (Pair (a, a))
      | Right -> None)
   | Gfm_dis gfm0 ->
     (match serialize mem gfm0 ast with
      | Some p ->
        let Pair (mem', msg') = p in
        let size = bytes_to_nat mem in
        (match Nat.eq_dec size (length msg') with
         | Left -> Some (Pair (mem', msg'))
         | Right -> None)
      | None -> None)
   | Gfm_dis_fixed (size, gfm0) ->
     (match serialize mem gfm0 ast with
      | Some p ->
        let Pair (mem', msg') = p in
        (match Nat.eq_dec size (length msg') with
         | Left -> Some (Pair (mem', msg'))
         | Right -> None)
      | None -> None)
   | Gfm_delim gfm0 ->
     (match serialize Nil gfm0 ast with
      | Some p ->
        let Pair (mem', msg') = p in
        (match substring byte_eq_dec mem (app msg' mem) with
         | Some p0 ->
           let Pair (_, l0) = p0 in
           (match l0 with
```

```
            | Nil -> Some (Pair (mem', (app msg' mem)))
            | Cons (_, _) -> None)
        | None -> None)
    | None -> None)
  | Gfm_fct fct0 -> fct0.fct_dir mem ast
  | _ -> None)
| Ast_sequence (a1, a2) ->
  (match gfm with
   | Gfm_dis gfm0 ->
     (match serialize mem gfm0 ast with
      | Some p ->
        let Pair (mem', msg') = p in
        let size = bytes_to_nat mem in
        (match Nat.eq_dec size (length msg') with
         | Left -> Some (Pair (mem', msg'))
         | Right -> None)
      | None -> None)
   | Gfm_dis_fixed (size, gfm0) ->
     (match serialize mem gfm0 ast with
      | Some p ->
        let Pair (mem', msg') = p in
        (match Nat.eq_dec size (length msg') with
         | Left -> Some (Pair (mem', msg'))
         | Right -> None)
      | None -> None)
   | Gfm_delim gfm0 ->
     (match serialize Nil gfm0 ast with
      | Some p ->
        let Pair (mem', msg') = p in
        (match substring byte_eq_dec mem (app msg' mem) with
         | Some p0 ->
           let Pair (_, l0) = p0 in
           (match l0 with
            | Nil -> Some (Pair (mem', (app msg' mem)))
            | Cons (_, _) -> None)
         | None -> None)
      | None -> None)
   | Gfm_sequence (g1, g2) ->
     (match serialize mem g1 a1 with
      | Some p ->
        let Pair (mem1, msg1) = p in
        (match serialize mem1 g2 a2 with
         | Some p0 ->
           let Pair (mem2, msg2) = p0 in
```

```
                Some (Pair (mem2, (app msg1 msg2)))
          | None -> None)
      | None -> None)
   | Gfm_fct fct0 -> fct0.fct_dir mem ast
   | _ -> None)
| Ast_repetition asts ->
  (match gfm with
   | Gfm_dis gfm0 ->
     (match serialize mem gfm0 ast with
      | Some p ->
        let Pair (mem', msg') = p in
        let size = bytes_to_nat mem in
        (match Nat.eq_dec size (length msg') with
         | Left -> Some (Pair (mem', msg'))
         | Right -> None)
      | None -> None)
   | Gfm_dis_fixed (size, gfm0) ->
     (match serialize mem gfm0 ast with
      | Some p ->
        let Pair (mem', msg') = p in
        (match Nat.eq_dec size (length msg') with
         | Left -> Some (Pair (mem', msg'))
         | Right -> None)
      | None -> None)
   | Gfm_delim gfm0 ->
     (match serialize Nil gfm0 ast with
      | Some p ->
        let Pair (mem', msg') = p in
        (match substring byte_eq_dec mem (app msg' mem) with
         | Some p0 ->
           let Pair (_, l0) = p0 in
           (match l0 with
            | Nil -> Some (Pair (mem', (app msg' mem)))
            | Cons (_, _) -> None)
         | None -> None)
      | None -> None)
   | Gfm_repetition gfm0 ->
     let num = bytes_to_nat mem in
     (match Nat.eq_dec num (length asts) with
      | Left ->
        (match serialize_repetition (serialize Nil gfm0) asts with
         | Some msg -> Some (Pair (Nil, msg))
         | None -> None)
      | Right -> None)
```

```
    | Gfm_fct fct0 -> fct0.fct_dir mem ast
    | _ -> None)
| Ast_star asts ->
  (match gfm with
    | Gfm_dis gfm0 ->
      (match serialize mem gfm0 ast with
       | Some p ->
         let Pair (mem', msg') = p in
         let size = bytes_to_nat mem in
         (match Nat.eq_dec size (length msg') with
          | Left -> Some (Pair (mem', msg'))
          | Right -> None)
       | None -> None)
    | Gfm_dis_fixed (size, gfm0) ->
      (match serialize mem gfm0 ast with
       | Some p ->
         let Pair (mem', msg') = p in
         (match Nat.eq_dec size (length msg') with
          | Left -> Some (Pair (mem', msg'))
          | Right -> None)
       | None -> None)
    | Gfm_delim gfm0 ->
      (match serialize Nil gfm0 ast with
       | Some p ->
         let Pair (mem', msg') = p in
         (match substring byte_eq_dec mem (app msg' mem) with
          | Some p0 ->
            let Pair (_, l0) = p0 in
            (match l0 with
             | Nil -> Some (Pair (mem', (app msg' mem)))
             | Cons (_, _) -> None)
          | None -> None)
       | None -> None)
    | Gfm_star gfm0 ->
      let num = bytes_to_nat mem in
      (match serialize_star (serialize Nil gfm0) asts with
       | Some msg ->
         (match Nat.eq_dec num (length msg) with
          | Left -> Some (Pair (Nil, msg))
          | Right -> None)
       | None -> None)
    | Gfm_fct fct0 -> fct0.fct_dir mem ast
    | _ -> None)
| Ast_choice a ->
```

```
(match gfm with
 | Gfm_dis gfm0 ->
   (match serialize mem gfm0 ast with
    | Some p ->
      let Pair (mem', msg') = p in
      let size = bytes_to_nat mem in
      (match Nat.eq_dec size (length msg') with
       | Left -> Some (Pair (mem', msg'))
       | Right -> None)
    | None -> None)
 | Gfm_dis_fixed (size, gfm0) ->
   (match serialize mem gfm0 ast with
    | Some p ->
      let Pair (mem', msg') = p in
      (match Nat.eq_dec size (length msg') with
       | Left -> Some (Pair (mem', msg'))
       | Right -> None)
    | None -> None)
 | Gfm_delim gfm0 ->
   (match serialize Nil gfm0 ast with
    | Some p ->
      let Pair (mem', msg') = p in
      (match substring byte_eq_dec mem (app msg' mem) with
       | Some p0 ->
         let Pair (_, l0) = p0 in
         (match l0 with
          | Nil -> Some (Pair (mem', (app msg' mem)))
          | Cons (_, _) -> None)
       | None -> None)
    | None -> None)
 | Gfm_choice (v, g1, g2) ->
   let w = bytes_to_nat mem in
   (match Nat.eq_dec v w with
    | Left -> serialize mem g1 a
    | Right -> serialize mem g2 a)
 | Gfm_fct fct0 -> fct0.fct_dir mem ast
 | _ -> None)
```

Annexe B

Spécification de protocoles dans le DSL

B.1 Protocole `TCP-Modbus`

L'ensemble des 16 messages `TCP-Modbus` implémentés pour les besoins d'évaluation de l'outil est donné en liste B.1. En pratique, il y a 8 types de messages questions/réponses.

Listing B.1 – Modèle pour le protocole TCP-Modbus (16 messages) exprimé dans notre DSL

```
Mfc01 := Fc Ref Coil
Fc := [Fixed,1] : E : Keyword,01
Ref := [Fixed,2] : E : Data
Coil := [Fixed,2] : E : Data

Rfc01 := Fc Count Coils
Fc := [Fixed,1] : E : Keyword,01
Count := [Fixed,1] : E : Counter
Coils := [Child] : T : Coil{Count}
Coil := [Fixed,1] : E : Data

Mfc02 := Fc Ref Inputs
Fc := [Fixed,1] : E : Keyword,02
Ref := [Fixed,2] : E : Data
Inputs := [Fixed,2] : E : Data

Rfc02 := Fc Count Inputs
Fc := [Fixed,1] : E : Keyword,02
Count := [Fixed,1] : E : Counter
Inputs := [Child] : T : Input{Count}
Input := [Fixed,1] : E : Data

Mfc03 := Fc Ref Count
Fc := [Fixed,1] : E : Keyword,03
Ref := [Fixed,2] : E : Data
Count := [Fixed,2] : E : Data
```

```
Rfc03 := Fc Count Regs
Fc := [Fixed,1] : E : Keyword,03
Count := [Fixed,1] : E : Counter
Regs := [Child] : T : Reg{Count}
Reg := [Fixed,2] : E : Data

Mfc04 := Fc Ref Count
Fc := [Fixed,1] : E : Keyword,04
Ref := [Fixed,2] : E : Data
Count := [Fixed,2] : E : Data

Rfc04 := Fc Count Regs
Fc := [Fixed,1] : E : Keyword,04
Count := [Fixed,1] : E : Counter
Regs := [Child] : T : Reg{Count}
Reg := [Fixed,2] : E : Data

Mfc05 := Fc Ref Coil Padd
Fc := [Fixed,1] : E : Keyword,05
Ref := [Fixed,2] : E : Data
Coil := [Fixed,1] : E : Data
Padd := [Fixed,1] : E : Keyword,00

Rfc05 := Fc Ref Coil Padd
Fc := [Fixed,1] : E : Keyword,05
Ref := [Fixed,2] : E : Data
Coil := [Fixed,1] : E : Data
Padd := [Fixed,1] : E : Keyword,00

Mfc06 := Fc Ref Reg
Fc := [Fixed,1] : E : Keyword,06
Ref := [Fixed,2] : E : Data
Reg := [Fixed,2] : E : Data

Rfc06 := Fc Ref Reg
Fc := [Fixed,1] : E : Keyword,06
Ref := [Fixed,2] : E : Data
Reg := [Fixed,2] : E : Data

Mfc16 := Fc Ref Count Regs
Fc := [Fixed,2] : E : Keyword,16
Count := [Fixed,2] : E : Counter
Ref := [Fixed,2] : E : Data
```

```
Regs := [Child] : T : Reg{Count}
Reg := [Fixed,2] : E : Data

Rfc16 := Fc Ref Count
Fc := [Fixed,1] : E : Keyword,16
Ref := [Fixed,2] : E : Data
Count := [Fixed,2] : E : Data

Mfc15 := Fc Ref BitCount ByteCount Coils
Fc := [Fixed,2] : E : Keyword,15
BitCount := [Fixed,2] : E : Data
Ref := [Fixed,2] : E : Data
ByteCount := [Fixed,1] : E : Length
Coils := [Length,{ByteCount}] : E : Data

Rfc15 := Fc Ref BitCount
Fc := [Fixed,1] : E : Keyword,15
Ref := [Fixed,2] : E : Data
BitCount := [Fixed,2] : E : Data
```

Annexe C

Détails sur les templates de génération de code

C.1 Exemple de référence pour la génération du code

Le message *M* est composé de trois sous-champs : *Fc*, *Count* et *Coils*. *Coils* est un tableau dont la taille est donnée par le champs *Count*, et contient un sous-champ *Elem* de type `Séquence`. *Elem* est composé de deux sous champs *S* et *D*.

Listing C.1 – Modèle pour le protocole `Running test` dans notre DSL

```
M := Fc Count Coils
Fc := [Fixed,1] : E : Keyword,10
Count := [Fixed,1] : E : Counter
Coils := [Child] : T : Coil{Count}
Coil := [Child] : L : S D
S := [Fixed,1] : E : Length
D := [Length,{S}] : E : Data
```

C.2 Les prototypes

```
bool Allocate_Message_M(Message_M_T** ppMessage);
bool Free_Message_M(Message_M_T* pMessage);
bool Parse_Message_M (Message_M_T** ppMessage, uint8_t* pData,
↪  uint32_t ui32DataSize);
bool GetSize_Message_M (Message_M_T* pMessage, uint32_t*
↪  pui32Size);
bool Serialize_Message_M (Message_M_T* pMessage, uint8_t* pData,
↪  uint32_t* pui32DataSize);
bool Initialize_Message_M (Message_M_T* pMessage);
bool Finalize_Message_M (Message_M_T* pMessage);
bool Message_M_get_Field_M_Fc (Message_M_T* pMessage,
↪  Field_M_Fc_T** ppField);
bool Message_M_set_Field_M_Fc (Message_M_T* pMessage, Field_M_Fc_T*
↪  pField);
bool Allocate_Field_M_Fc (Field_M_Fc_T** ppField);
bool Free_Field_M_Fc (Message_M_T* pMessage, Field_M_Fc_T* pField);
```

```
bool Initialize_Field_M_Fc (Message_M_T* pMessage, Field_M_Fc_T*
↪  pField);
bool Finalize_Field_M_Fc (Message_M_T* pMessage, Field_M_Fc_T*
↪  pField);
bool SetKeywordI_Field_M_Fc (Message_M_T* pMessage, Field_M_Fc_T*
↪  pField, uint32_t ui32Keyword);
bool GetKeywordI_Field_M_Fc (Message_M_T* pMessage, Field_M_Fc_T*
↪  pField, uint32_t* pui32Keyword);
bool Message_M_get_Field_M_Count (Message_M_T* pMessage,
↪  Field_M_Count_T** ppField);
bool Message_M_set_Field_M_Count (Message_M_T* pMessage,
↪  Field_M_Count_T* pField);
bool Allocate_Field_M_Count (Field_M_Count_T** ppField);
bool Free_Field_M_Count (Message_M_T* pMessage, Field_M_Count_T*
↪  pField);
bool Initialize_Field_M_Count (Message_M_T* pMessage,
↪  Field_M_Count_T* pField);
bool Finalize_Field_M_Count (Message_M_T* pMessage,
↪  Field_M_Count_T* pField);
bool SetCounter_Field_M_Count (Message_M_T* pMessage,
↪  Field_M_Count_T* pField, uint32_t ui32Counter);
bool GetCounter_Field_M_Count (Message_M_T* pMessage,
↪  Field_M_Count_T* pField, uint32_t* pui32Counter);
bool Message_M_get_Field_M_Coils (Message_M_T* pMessage,
↪  Field_M_Coils_T** ppField);
bool Message_M_set_Field_M_Coils (Message_M_T* pMessage,
↪  Field_M_Coils_T* pField);
bool Allocate_Field_M_Coils (Field_M_Coils_T** ppField);
bool Free_Field_M_Coils (Message_M_T* pMessage, Field_M_Coils_T*
↪  pField);
bool Initialize_Field_M_Coils (Message_M_T* pMessage,
↪  Field_M_Coils_T* pField);
bool Finalize_Field_M_Coils (Message_M_T* pMessage,
↪  Field_M_Coils_T* pField);
bool GetTabLen_Field_M_Coils (Message_M_T* pMessage,
↪  Field_M_Coils_T* pField, uint32_t* pTabLen);
bool InitTab_Field_M_Coils (Message_M_T* pMessage, Field_M_Coils_T*
↪  pField);
bool Field_M_Coils_GetTabField_Field_M_Coil (Message_M_T* pMessage,
↪  Field_M_Coils_T* pParentField, Field_M_Coil_T** ppField,
↪  uint32_t ui32Index);
bool Field_M_Coils_SetTabField_Field_M_Coil (Message_M_T* pMessage,
↪  Field_M_Coils_T* pParentField, Field_M_Coil_T* pField, uint32_t
↪  ui32Index);
```

```
bool Allocate_Field_M_Coil (Field_M_Coil_T** ppField);
bool Free_Field_M_Coil (Message_M_T* pMessage, Field_M_Coil_T*
↪  pField);
bool Initialize_Field_M_Coil (Message_M_T* pMessage,
↪  Field_M_Coil_T* pField);
bool Finalize_Field_M_Coil (Message_M_T* pMessage, Field_M_Coil_T*
↪  pField);
bool Field_M_Coil_Get_Field_M_S (Message_M_T* pMessage,
↪  Field_M_Coil_T* pParentField, Field_M_S_T** ppField);
bool Field_M_Coil_Set_Field_M_S (Message_M_T* pMessage,
↪  Field_M_Coil_T* pParentField, Field_M_S_T* pField);
bool Allocate_Field_M_S (Field_M_S_T** ppField);
bool Free_Field_M_S (Message_M_T* pMessage, Field_M_S_T* pField);
bool Initialize_Field_M_S (Message_M_T* pMessage, Field_M_S_T*
↪  pField);
bool Finalize_Field_M_S (Message_M_T* pMessage, Field_M_S_T*
↪  pField);
bool SetLength_Field_M_S (Message_M_T* pMessage, Field_M_S_T*
↪  pField, uint32_t ui32Length);
bool GetLength_Field_M_S (Message_M_T* pMessage, Field_M_S_T*
↪  pField, uint32_t* pui32Length);
bool Field_M_Coil_Get_Field_M_D (Message_M_T* pMessage,
↪  Field_M_Coil_T* pParentField, Field_M_D_T** ppField);
bool Field_M_Coil_Set_Field_M_D (Message_M_T* pMessage,
↪  Field_M_Coil_T* pParentField, Field_M_D_T* pField);
bool Allocate_Field_M_D (Field_M_D_T** ppField);
bool Free_Field_M_D (Message_M_T* pMessage, Field_M_D_T* pField);
bool Initialize_Field_M_D (Message_M_T* pMessage, Field_M_D_T*
↪  pField);
bool Finalize_Field_M_D (Message_M_T* pMessage, Field_M_D_T*
↪  pField);
bool SetData_Field_M_D (Message_M_T* pMessage, Field_M_D_T* pField,
↪  uint8_t* pData, uint32_t ui32DataSize);
bool GetData_Field_M_D (Message_M_T* pMessage, Field_M_D_T* pField,
↪  uint8_t** ppData, uint32_t* pui32DataSize);
bool Parse_Field_M_Fc (Message_M_T* pMessage, uint8_t* pData,
↪  uint32_t ui32DataSize, uint32_t* pNbRead);
bool Serialize_Field_M_Fc (Message_M_T* pMessage, uint8_t* pData,
↪  uint32_t ui32DataSize, uint32_t* pui32NbWrite);
bool GetSize_Field_M_Fc (Message_M_T* pMessage, uint32_t*
↪  pui32Size);
bool Parse_Field_M_Count (Message_M_T* pMessage, uint8_t* pData,
↪  uint32_t ui32DataSize, uint32_t* pNbRead);
```

```
bool Serialize_Field_M_Count (Message_M_T* pMessage, uint8_t*
↪  pData, uint32_t ui32DataSize, uint32_t* pui32NbWrite);
bool GetSize_Field_M_Count (Message_M_T* pMessage, uint32_t*
↪  pui32Size);
bool Parse_Field_M_Coils (Message_M_T* pMessage, uint8_t* pData,
↪  uint32_t ui32DataSize, uint32_t* pNbRead);
bool Serialize_Field_M_Coils (Message_M_T* pMessage, uint8_t*
↪  pData, uint32_t ui32DataSize, uint32_t* pui32NbWrite);
bool GetSize_Field_M_Coils (Message_M_T* pMessage, uint32_t*
↪  pui32Size);
bool Parse_Field_M_Coil (Message_M_T* pMessage, uint8_t* pData,
↪  uint32_t ui32DataSize, uint32_t* pNbRead);
bool Serialize_Field_M_Coil (Message_M_T* pMessage, uint8_t* pData,
↪  uint32_t ui32DataSize, uint32_t* pui32NbWrite);
bool GetSize_Field_M_Coil (Message_M_T* pMessage, uint32_t*
↪  pui32Size);
bool Parse_Field_M_S (Message_M_T* pMessage, uint8_t* pData,
↪  uint32_t ui32DataSize, uint32_t* pNbRead);
bool Serialize_Field_M_S (Message_M_T* pMessage, uint8_t* pData,
↪  uint32_t ui32DataSize, uint32_t* pui32NbWrite);
bool GetSize_Field_M_S (Message_M_T* pMessage, uint32_t*
↪  pui32Size);
bool Parse_Field_M_D (Message_M_T* pMessage, uint8_t* pData,
↪  uint32_t ui32DataSize, uint32_t* pNbRead);
bool Serialize_Field_M_D (Message_M_T* pMessage, uint8_t* pData,
↪  uint32_t ui32DataSize, uint32_t* pui32NbWrite);
bool GetSize_Field_M_D (Message_M_T* pMessage, uint32_t*
↪  pui32Size);
```

C.3 Structures internes

Ensemble des structures internes générées pour pouvoir parser et sérialiser le message. Les fonctions accesseurs manipulent cette structure interne.

```
typedef struct Message_M_T_ {
  struct Field_M_Fc_T_*  pField_M_Fc;
  struct Field_M_Count_T_*  pField_M_Count;
  struct Field_M_Coils_T_*  pField_M_Coils;
} Message_M_T;

typedef struct Field_M_Fc_T_ {
  uint32_t  ui32ElementTypeKeywordValue;
  bool  bIsDefined;
```

```
} Field_M_Fc_T;

typedef struct Field_M_Count_T_ {
  uint32_t  ui32Counter;
  bool  bIsDefined;
} Field_M_Count_T;

typedef struct Field_M_Coils_T_ {
  struct Field_M_Coil_T_**  pTabFieldElement;
  uint32_t  ui32Index;
} Field_M_Coils_T;

typedef struct Field_M_Coil_T_ {
  struct Field_M_S_T_*  pField_M_S;
  struct Field_M_D_T_*  pField_M_D;
} Field_M_Coil_T;

typedef struct Field_M_S_T_ {
  uint32_t  ui32Length;
  bool  bIsDefined;
} Field_M_S_T;

typedef struct Field_M_D_T_ {
  uint8_t*  pData;
  uint32_t  ui32DataSize;
  bool  bIsDefined;
} Field_M_D_T;
```

C.4 Code

C.4.1 Allocate et Free

Code pour les fonctions `Alloc` et `Free` pour le "message" et pour un nœud terminal Fc.

```
bool Allocate_Message_M(Message_M_T** ppMessage) {
  bool res = false;
  Message_M_T* pMessage = NULL;

  if (ppMessage == NULL)     goto end;
  MALLOC_STRUCT(Message_M_T, pMessage);
  (*ppMessage) = pMessage;
  res = true;
end:
```

```
  return (res);
 }
bool Free_Message_M(Message_M_T* pMessage) {
  bool res = false;

  if (pMessage == NULL)     goto end;
  res = Free_Field_M_Coils(pMessage, pMessage->pField_M_Coils);
  if (!res) {
    goto end;
  }// end if
  res = Free_Field_M_Count(pMessage, pMessage->pField_M_Count);
  if (!res) {
    goto end;
  }// end if
  res = Free_Field_M_Fc(pMessage, pMessage->pField_M_Fc);
  if (!res) {
    goto end;
  }// end if
  FREE(pMessage);
  res = true;
end:
  return (res);
 }

 ...

 bool Allocate_Field_M_Fc (Field_M_Fc_T** ppField) {
  bool     res = false;
  Field_M_Fc_T* pField = NULL;

  // check entries
  if (ppField == NULL)    goto end;

  MALLOC_STRUCT(Field_M_Fc_T, pField);
  pField->ui32ElementTypeKeywordValue = 10;
  res = true;
  (*ppField) = pField;
end:
  return (res);
 }
bool Free_Field_M_Fc (Message_M_T* pMessage, Field_M_Fc_T* pField)
↪  {
  bool     res = false;
```

```
  // check entries
  // pField can be null in the case of optional fields
  if (pField == NULL)    goto end;

  res = true;
  FREE(pField);
end:
  return (res);
 }
```

C.4.2 Initialize et Finalize

Code pour les fonctions `Initialize` et `Finalize` pour le "message" et pour un nœud terminal `S` sur lequel des obfuscations ont été appliquées.

```
bool Initialize_Message_M (Message_M_T* pMessage) {
  bool res = false;

  if (pMessage == NULL)     goto end;
  res = Initialize_Field_M_Fc(pMessage, pMessage->pField_M_Fc);
  if (!res) {
    goto end;
  }// end if
  res = Initialize_Field_M_Count(pMessage,
  ↪  pMessage->pField_M_Count);
  if (!res) {
    goto end;
  }// end if
  res = Initialize_Field_M_Coils(pMessage,
  ↪  pMessage->pField_M_Coils);
  if (!res) {
    goto end;
  }// end if
  res = true;
end:
  return (res);
 }
bool Finalize_Message_M (Message_M_T* pMessage) {
  bool res = false;

  if (pMessage == NULL)     goto end;
  res = Finalize_Field_M_Fc(pMessage, pMessage->pField_M_Fc);
  if (!res) {
    goto end;
```

```
  }// end if
  res = Finalize_Field_M_Count(pMessage, pMessage->pField_M_Count);
  if (!res) {
    goto end;
  }// end if
  res = Finalize_Field_M_Coils(pMessage, pMessage->pField_M_Coils);
  if (!res) {
    goto end;
  }// end if
  res = true;
end:
  return (res);
 }

...

bool Finalize_Field_M_S (Message_M_T* pMessage, Field_M_S_T*
↪  pField) {
  bool res = false;
  Field_M_S_E532466E_T*    pField_M_S_E532466E = NULL;
  Field_M_S_3A1D75A0_T*    pField_M_S_3A1D75A0 = NULL;
  uint8_t*    pDataL          = NULL;
  uint32_t    ui32DataSizeL   = 0;
  uint8_t*    pDataR          = NULL;
  uint32_t    ui32DataSizeR   = 0;
  uint32_t    i               = 0;
  uint32_t    ui32DataSize    = 0;
  res = Field_M_S_Get_Field_M_S_E532466E(pMessage, pField,
  ↪  &pField_M_S_E532466E);
  if (!res)   goto end;
  res = Field_M_S_Get_Field_M_S_3A1D75A0(pMessage, pField,
  ↪  &pField_M_S_3A1D75A0);
  if (!res)   goto end;
  res = GetData_Field_M_S_E532466E(pMessage, pField_M_S_E532466E,
  ↪  &pDataL, &ui32DataSizeL);
  if (!res) {
    FREE(pDataL);
    goto end;
  } // end if
  res = GetData_Field_M_S_3A1D75A0(pMessage, pField_M_S_3A1D75A0,
  ↪  &pDataR, &ui32DataSizeR);
  if (!res) {
    FREE(pDataL);
    FREE(pDataR);
```

```
    goto end;
  } // end if
  ui32DataSize = ui32DataSizeL;
  uint32_t    ui32Len    = 0;
  for (i = 0; i < ui32DataSize; i++) {
    ((uint8_t*) &ui32Len)[i] = pDataL[i] + pDataR[i];
  }// end for
  FREE(pDataL);
  FREE(pDataR);
  res = SetLength_Field_M_S(pMessage, pField, ui32Len);
  if (!res)   goto end;
  pField->bIsDefined = true;
  res = true;
end:
  return (res);
 }
bool Initialize_Field_M_S (Message_M_T* pMessage, Field_M_S_T*
↪  pField) {
  bool res = false;
  pField->bIsDefined = true;
  Field_M_S_E532466E_T*    pField_M_S_E532466E = NULL;
  Field_M_S_3A1D75A0_T*    pField_M_S_3A1D75A0 = NULL;
  uint32_t    ui32DataSize    = 0;
  uint8_t*    pDataL          = NULL;
  uint32_t    ui32DataSizeL   = 0;
  uint8_t*    pDataR          = NULL;
  uint32_t    ui32DataSizeR   = 0;
  uint32_t    i               = 0;
  uint8_t     ui8Random       = 0;
  ui32DataSize = 1;
  ui32DataSizeL = ui32DataSize;
  ui32DataSizeR = ui32DataSize;
  MALLOC_ARRAY(ui32DataSizeL, pDataL);
  MALLOC_ARRAY(ui32DataSizeR, pDataR);
  res = Allocate_Field_M_S_E532466E(&pField_M_S_E532466E);
  if (!res)   goto end;
  res = Allocate_Field_M_S_3A1D75A0(&pField_M_S_3A1D75A0);
  if (!res)   goto end;
  res = Field_M_S_Set_Field_M_S_E532466E(pMessage, pField,
  ↪  pField_M_S_E532466E);
  if (!res)   goto end;
  res = Field_M_S_Set_Field_M_S_3A1D75A0(pMessage, pField,
  ↪  pField_M_S_3A1D75A0);
  if (!res)   goto end;
```

```
  uint32_t    ui32Len    = 0;
  res = GetLength_Field_M_S(pMessage, pField, &ui32Len);
  if (!res)   goto end;
  for (i = 0; i < ui32DataSize; i++) {
    ui8Random = (uint8_t) getAleaValue(0x100);
    pDataL[i] = ((uint8_t*) &ui32Len)[i] - ui8Random;
    pDataR[i] = ui8Random;
  }// end for
  res = SetData_Field_M_S_E532466E(pMessage, pField_M_S_E532466E,
  ↪  pDataL, ui32DataSizeL);
  if (!res) {
    FREE(pDataL);
    FREE(pDataR);
    goto end;
  } // end if
  res = SetData_Field_M_S_3A1D75A0(pMessage, pField_M_S_3A1D75A0,
  ↪  pDataR, ui32DataSizeR);
  if (!res) {
    FREE(pDataL);
    FREE(pDataR);
    goto end;
  } // end if
    FREE(pDataL);
    FREE(pDataR);
  pDataL = NULL;
  pDataR = NULL;
  res = true;
end:
  return (res);
 }
```

C.4.3 Parse, Serialize et GetSize

Code pour les fonctions `Parse`, `Serialize` et `GetSize` pour le "message".

```
bool Parse_Message_M (Message_M_T** ppMessage, uint8_t* pData,
↪  uint32_t ui32DataSize) {
  uint32_t    ui32DataOffset = 0; // would be used in the case of
  ↪  reversed parsing obfuscation
  uint32_t    ui32DataOffsetBegin = 0;
  uint32_t    ui32NbRead = 0;
  uint32_t    ui32RemainingSize = ui32DataSize;
  bool        res = false;
  Message_M_T*    pMessage = NULL;
```

```
  if (ppMessage == NULL)      goto end;
  if (pData == NULL)          goto end;
  if (ui32DataSize == 0)      goto end;

  res = Allocate_Message_M(&pMessage);
  if (!res)     goto end;
  ui32DataOffset = ui32DataOffsetBegin;
  res = Parse_Field_M_Fc(pMessage, pData + ui32DataOffset ,
  ↪  ui32RemainingSize, &ui32NbRead);
  if (!res) {
    goto end;
  }// end if
  ui32RemainingSize -= ui32NbRead;
  ui32DataOffsetBegin += ui32NbRead;
  ui32DataOffset = ui32DataOffsetBegin;
  res = Parse_Field_M_Count(pMessage, pData + ui32DataOffset ,
  ↪  ui32RemainingSize, &ui32NbRead);
  if (!res) {
    goto end;
  }// end if
  ui32RemainingSize -= ui32NbRead;
  ui32DataOffsetBegin += ui32NbRead;
  ui32DataOffset = ui32DataOffsetBegin;
  res = Parse_Field_M_Coils(pMessage, pData + ui32DataOffset ,
  ↪  ui32RemainingSize, &ui32NbRead);
  if (!res) {
    goto end;
  }// end if
  ui32RemainingSize -= ui32NbRead;
  ui32DataOffsetBegin += ui32NbRead;
  res = Finalize_Message_M (pMessage);
  if (!res)   goto end;
  res = true;
  (*ppMessage) = pMessage;
end:
  if (!res) {
    // carefull, will have a memory leak here
    // error, so call cleanup cleanup function
    Free_Message_M(pMessage);
  }// end if
  return (res);
```

```
 }
// if size is too small, required size is returned in pui32DataSize
bool Serialize_Message_M (Message_M_T* pMessage, uint8_t* pData,
↪  uint32_t* pui32DataSize) {
  uint32_t    ui32OffsetBegin         = 0;
  uint32_t    ui32Offset              = 0;
  uint32_t    ui32FieldSize           = 0;
  uint32_t    ui32RequiredDataSize    = 0;
  uint32_t    ui32RemainingDataSize   = 0;
  uint32_t    ui32DataSize            = 0;
  bool        res                     = false;
  uint32_t    ui32NbWrite             = 0;

  if (pData == NULL)          goto end;
  if (pMessage == NULL)       goto end;
  if (pui32DataSize == NULL)  goto end;
  ui32DataSize = (*pui32DataSize);
  res = Initialize_Message_M (pMessage);
  if (!res)   goto end;
  res = GetSize_Message_M (pMessage, &ui32RequiredDataSize);
  if (!res)   goto end;
  if (ui32RequiredDataSize > ui32DataSize) {
    // error, not enough space for serialization
    printf("error, not enough space for serialization of message
    ↪  (M)\n");
    (*pui32DataSize) = ui32RequiredDataSize;
    goto end;
  }

  ui32RemainingDataSize = ui32DataSize;
  ui32Offset = ui32OffsetBegin;
  // no check on size because allready taken into acount by
  ↪  Message_GetSize
  res = Serialize_Field_M_Fc (pMessage, pData + ui32Offset,
  ↪  ui32RemainingDataSize, &ui32NbWrite);
  if (!res) {
    printf("error, cannot serialize field Fc of message M\n");
    goto end;
  }// end if
  ui32RemainingDataSize -= ui32NbWrite;
  ui32OffsetBegin += ui32NbWrite;
  ui32Offset = ui32OffsetBegin;
  // no check on size because allready taken into acount by
  ↪  Message_GetSize
```

```
  res = Serialize_Field_M_Count (pMessage, pData + ui32Offset,
  ↪  ui32RemainingDataSize, &ui32NbWrite);
  if (!res) {
    printf("error, cannot serialize field Count of message M\n");
    goto end;
  }// end if
  ui32RemainingDataSize -= ui32NbWrite;
  ui32OffsetBegin += ui32NbWrite;
  ui32Offset = ui32OffsetBegin;
  // no check on size because allready taken into acount by
  ↪  Message_GetSize
  res = Serialize_Field_M_Coils (pMessage, pData + ui32Offset,
  ↪  ui32RemainingDataSize, &ui32NbWrite);
  if (!res) {
    printf("error, cannot serialize field Coils of message M\n");
    goto end;
  }// end if
  ui32RemainingDataSize -= ui32NbWrite;
  ui32OffsetBegin += ui32NbWrite;
  res = true;
end:
  return (res);
 }
bool GetSize_Message_M (Message_M_T* pMessage, uint32_t* pui32Size)
↪  {
  bool        res = false;
  uint32_t    ui32GetFieldSize = 0;
  uint32_t    ui32ElementSize = 0;

  if (pMessage == NULL)    goto end;
  if (pui32Size == NULL)   goto end;
  res = Initialize_Message_M (pMessage);
  if (!res)   goto end;
  res = GetSize_Field_M_Fc (pMessage, &ui32GetFieldSize);
  if (!res) {
    printf("error, cannot get size of field Fc of message M\n");
    goto end;
    }
  ui32ElementSize += ui32GetFieldSize;
  res = GetSize_Field_M_Count (pMessage, &ui32GetFieldSize);
  if (!res) {
    printf("error, cannot get size of field Count of message M\n");
    goto end;
    }
```

```
  ui32ElementSize += ui32GetFieldSize;
  res = GetSize_Field_M_Coils (pMessage, &ui32GetFieldSize);
  if (!res) {
    printf("error, cannot get size of field Coils of message M\n");
    goto end;
    }
  ui32ElementSize += ui32GetFieldSize;
  (*pui32Size) = ui32ElementSize;
  res = true;
end:
  return (res);
 }
```

C.4.4 accesseurs directs

Code pour les fonctions accesseurs directs utilisées par les fonctions `Parse`, `Serialize` et `GetSize` pour les nœuds S et D. Ces fonctions sont destinées à n'être utilisées qu'en interne.

```
bool Message_M_get_Field_M_S (Message_M_T* pMessage, Field_M_S_T**
↪  ppField) {
  bool res = false;

  if (pMessage == NULL)   goto end;
  if (ppField == NULL)   goto end;
  if (pMessage->pField_M_Coils == NULL)   goto end;
  if (pMessage->pField_M_Coils->pTabFieldElement[pMessage->
  ↪  pField_M_Coils->ui32Index] == NULL)   goto
  ↪  end;
  if (pMessage->pField_M_Coils->pTabFieldElement[pMessage->
  ↪  pField_M_Coils->ui32Index]->pField_M_S == NULL)
  ↪  {
    res = Allocate_Field_M_S
    ↪  (&(pMessage->pField_M_Coils->pTabFieldElement[pMessage->
    ↪  pField_M_Coils->ui32Index]->pField_M_S));
    if (!res)   goto end;
  }// end if
  (*ppField) =
  ↪  pMessage->pField_M_Coils->pTabFieldElement[pMessage->
  ↪  pField_M_Coils->ui32Index]->pField_M_S;
  res = true;
end:
  return (res);
 }
```

```
bool Message_M_set_Field_M_S (Message_M_T* pMessage, Field_M_S_T*
↪  pField) {
  bool res = false;

  if (pMessage == NULL)   goto end;
  if (pField == NULL)   goto end;
  if (pMessage->pField_M_Coils == NULL) {
    Field_M_Coils_T* pField_M_Coils  = NULL;
    res = Allocate_Field_M_Coils (&pField_M_Coils);
    if ( !res )   goto end;
    res = Message_M_set_Field_M_Coils (pMessage, pField_M_Coils);
  }// end if Coils == NULL
  if (pMessage->pField_M_Coils->pTabFieldElement[pMessage->」
  ↪  pField_M_Coils->ui32Index] == NULL)
  ↪  {
    Field_M_Coil_T* pField_M_Coil  = NULL;
    res = Allocate_Field_M_Coil (&pField_M_Coil);
    if ( !res )   goto end;
    res = Field_M_Coils_SetTabField_Field_M_Coil (pMessage,
    ↪  pMessage->pField_M_Coils, pField_M_Coil,
    ↪  pMessage->pField_M_Coils->ui32Index);
  }// end if Coil == NULL
  if (pMessage->pField_M_Coils->pTabFieldElement[pMessage->」
  ↪  pField_M_Coils->ui32Index]->pField_M_S != NULL)
  ↪  {
    res = Free_Field_M_S(pMessage,
    ↪  pMessage->pField_M_Coils->pTabFieldElement[pMessage->」
    ↪  pField_M_Coils->ui32Index]->pField_M_S);
    if (!res)     goto end;
  }
  pMessage->pField_M_Coils->pTabFieldElement[pMessage->」
  ↪  pField_M_Coils->ui32Index]->pField_M_S =
  ↪  pField;
  res = true;
end:
  return (res);
 }

bool Message_M_get_Field_M_D (Message_M_T* pMessage, Field_M_D_T**
↪  ppField) {
  bool res = false;

  if (pMessage == NULL)   goto end;
  if (ppField == NULL)   goto end;
```

```
  if (pMessage->pField_M_Coils == NULL)   goto end;
  if (pMessage->pField_M_Coils->pTabFieldElement[pMessage->
  ↪  pField_M_Coils->ui32Index] == NULL)   goto
  ↪  end;
  if (pMessage->pField_M_Coils->pTabFieldElement[pMessage->
  ↪  pField_M_Coils->ui32Index]->pField_M_D == NULL)   goto
  ↪  end;
  (*ppField) =
  ↪  pMessage->pField_M_Coils->pTabFieldElement[pMessage->
  ↪  pField_M_Coils->ui32Index]->pField_M_D;
  res = true;
end:
  return (res);
 }
bool Message_M_set_Field_M_D (Message_M_T* pMessage, Field_M_D_T*
↪  pField) {
  bool res = false;

  if (pMessage == NULL)   goto end;
  if (pField == NULL)   goto end;
  if (pMessage->pField_M_Coils == NULL) {
    Field_M_Coils_T* pField_M_Coils  = NULL;
    res = Allocate_Field_M_Coils (&pField_M_Coils);
    if ( !res )   goto end;
    res = Message_M_set_Field_M_Coils (pMessage, pField_M_Coils);
  }// end if Coils == NULL
  if (pMessage->pField_M_Coils->pTabFieldElement[pMessage->
  ↪  pField_M_Coils->ui32Index] == NULL)
  ↪  {
    Field_M_Coil_T* pField_M_Coil  = NULL;
    res = Allocate_Field_M_Coil (&pField_M_Coil);
    if ( !res )   goto end;
    res = Field_M_Coils_SetTabField_Field_M_Coil (pMessage,
    ↪  pMessage->pField_M_Coils, pField_M_Coil,
    ↪  pMessage->pField_M_Coils->ui32Index);
  }// end if Coil == NULL
  if (pMessage->pField_M_Coils->pTabFieldElement[pMessage->
  ↪  pField_M_Coils->ui32Index]->pField_M_D != NULL)
  ↪  {
    res = Free_Field_M_D(pMessage,
    ↪  pMessage->pField_M_Coils->pTabFieldElement[pMessage->
    ↪  pField_M_Coils->ui32Index]->pField_M_D);
    if (!res)     goto end;
  }
```

```
  pMessage->pField_M_Coils->pTabFieldElement[pMessage->
   ↪  pField_M_Coils->ui32Index]->pField_M_D =
   ↪  pField;
  res = true;
end:
  return (res);
 }
```

C.4.5 CMake

Le code *Cmake* permettant de créer automatiquement un projet pour le protocol `test_running_example`.

```
project(Test_RunningExample)
cmake_minimum_required(VERSION 2.8)

include_directories(
  ${all_include_directories}
  ${CMAKE_CURRENT_SOURCE_DIR})

set(generated_files
  ${CMAKE_BINARY_DIR}/ProtoObf_test_running_example.c
  ${CMAKE_BINARY_DIR}/ProtoObf_test_running_example.h
  ${CMAKE_BINARY_DIR}/prototypes_test_running_example.h
  )

add_custom_command(
    OUTPUT
        ${generated_files}
    COMMAND
        ${grammar_parser_binary_path}
         ↪  "${CMAKE_CURRENT_SOURCE_DIR}/test_running_example.txt"
         ↪  "${NB_OBFUSCATION}" "${SEED}" "${LOG}" >
         ↪  ${CMAKE_CURRENT_BINARY_DIR}/tmp_output.log
    VERBATIM)

add_library(
    generate-src_running_example ${generated_files})


set(Test_RunningExample_files
        CMakeLists.txt
        main.c
        test_running_example.c
```

```
        test_running_example.h
        ${common_files}
  test_running_example.txt
  )

add_executable(Test_RunningExample ${Test_RunningExample_files})

target_link_libraries(Test_RunningExample
↪  generate-src_running_example)
```

C.4.6 Project code

Code de test pour vérifier le bon fonctionnement, le nombre d'allocations effectuées et la vitesse d'execution du parsing et de la sérialisation.

```
bool test_simple_serialization(FILE* pLogFile)
{
    bool                res                 = false;
    uint8_t             i;
    Message_M_T*        pM;
    Message_M_T*        pMparsed;
    Field_M_Fc_T*       pFc;
    Field_M_Count_T*    pCount;
    Field_M_Coils_T*    pCoils;
    Field_M_Coil_T*     pCoil;
    Field_M_S_T*        pS;
    Field_M_D_T*        pD;
    uint32_t            serializedSize;
    uint8_t*            pSerializedBuffer;
    FILE*               pOutputFile;
    struct timespec end, start;
    uint64_t delta_us;
    uint64_t    nbAllocs;
    uint64_t    nbTotalAllocs;
    uint8_t     ui8BufferSize;
    uint8_t     ui8BufferOffset;

    uint8_t ui8TableSize = (uint8_t) getAleaValue(0x10);

    fprintf(pLogFile, " Running example\n");

    for (i = 0; i < sizeof(buffer); i++)
    {
        buffer[i] = (uint8_t) getAleaValue(0);
```

```
    }

    res = Allocate_Message_M(&pM);
    if (!res) goto end;
    res = Allocate_Field_M_Fc(&pFc);
    if (!res) goto end;
    res = Allocate_Field_M_Count(&pCount);
    if (!res) goto end;


    res = Message_M_set_Field_M_Fc(pM, pFc);
    if (!res) goto end;
    res = SetCounter_Field_M_Count(pM, pCount, ui8TableSize);
    if (!res) goto end;
    res = Message_M_set_Field_M_Count(pM, pCount);
    if (!res) goto end;

    res = Allocate_Field_M_Coils(&pCoils);
    if (!res) goto end;
    res = Message_M_set_Field_M_Coils(pM, pCoils);
    if (!res) goto end;

    res = InitTab_Field_M_Coils(pM, pCoils);
    if (!res) goto end;

    pCoils->ui32Index = 0;

    for (i = 0; i < ui8TableSize; i++)
    {
        res = Allocate_Field_M_Coil(&pCoil);
        if (!res) goto end;
        res = Allocate_Field_M_S(&pS);
        if (!res) goto end;
        res = Allocate_Field_M_D(&pD);
        if (!res) goto end;

        res = Field_M_Coils_SetTabField_Field_M_Coil(pM, pCoils,
        ↪  pCoil, i);
        if (!res) goto end;

        res = Field_M_Coil_Set_Field_M_S(pM, pCoil, pS);
        if (!res) goto end;
        res = Field_M_Coil_Set_Field_M_D(pM, pCoil, pD);
        if (!res) goto end;
```

```
    ui8BufferSize = getAleaValue(sizeof(buffer)/2);
    ui8BufferOffset = sizeof(buffer) - ui8BufferSize;
    ui8BufferOffset -= getAleaValue(ui8BufferOffset);
    //printf("buffer (%d) size (%X) offset (%X)\n", i,
    ↪  ui8BufferSize, ui8BufferOffset);
    res = SetData_Field_M_D(pM, pD, buffer + ui8BufferOffset,
    ↪  ui8BufferSize);
    if (!res) goto end;
}


res = GetSize_Message_M(pM, &serializedSize);
if (!res) goto end;
fprintf(pLogFile, "serialized size : %08X\n", serializedSize);
// should be 0x43 with this implem (no obfuscations)


clock_gettime(CLOCK_MONOTONIC_RAW, &start);

MALLOC_ARRAY(serializedSize, pSerializedBuffer);
res = Serialize_Message_M (pM, pSerializedBuffer,
↪  &serializedSize);
if (!res)
{
    printf("error in Serialize_Message_M\n");
    return false;
}
clock_gettime(CLOCK_MONOTONIC_RAW, &end);
delta_us = (end.tv_sec - start.tv_sec) * 1000000 + (end.tv_nsec
↪  - start.tv_nsec) / 1000;
getAllocations(&nbTotalAllocs, &nbAllocs);

// log
fprintf(pLogFile, "serialization time: %lu\n", delta_us);
fprintf(pLogFile, "serialization allocations : %08X\n",
↪  nbTotalAllocs);

// save
pOutputFile = fopen("simple.bin", "w");
fwrite(pSerializedBuffer, 1, serializedSize, pOutputFile);
fclose(pOutputFile);
```

```
    // reset and start logging for parsing
    resetNbAllocations();
    clock_gettime(CLOCK_MONOTONIC_RAW, &start);
    res = Parse_Message_M(&pMparsed, pSerializedBuffer,
    ↪  serializedSize);
    if (!res)
    {
        printf("Error, MUST WORK (res is false)\n");
        goto end;
    }
    if (pMparsed == NULL)
    {
        printf("Error, MUST WORK\n");
        goto end;
    }

    clock_gettime(CLOCK_MONOTONIC_RAW, &end);
    delta_us = (end.tv_sec - start.tv_sec) * 1000000 + (end.tv_nsec
    ↪  - start.tv_nsec) / 1000;
    getAllocations(&nbTotalAllocs, &nbAllocs);
    // log
    fprintf(pLogFile, "parsing time: %lu\n", delta_us);
    fprintf(pLogFile, "parsing allocations : %08X\n",
    ↪  nbTotalAllocs);

    // check validity of elements
    res = true;
end:
    if (!res)
        printf("Error in test\n");
    return (res);
}
```

Bibliographie

[Angluin 1987] Dana Angluin. *Learning Regular Sets from Queries and Counterexamples*. Information and Computation, vol. 75, no. 2, pages 87–106, 1987. (Cité en pages 14 et 23.)

[Antunes 2011] J. Antunes, N. Neves et P. Verissimo. *Reverse Engineering of Protocols from Network Traces*. Dans 2011 18th Working Conference on Reverse Engineering (WCRE), pages 169–178, New York, NY, USA, 2011. IEEE. (Cité en pages 12 et 14.)

[Apache 2017] Apache. *Apache Thrift*. `https://thrift.apache.org/`, 2017. (Cité en page 76.)

[asn 2017] *ASN.1 Made Simple*. `http://www.oss.com/asn1/resources/asn1-made-simple/introduction.html`, 2017. (Cité en pages 76 et 78.)

[Banescu 2016] Sebastian Banescu, Christian Collberg, Vijay Ganesh, Zack Newsham et Alexander Pretschner. *Code obfuscation against symbolic execution attacks*. Dans Proceedings of the 32nd Annual Conference on Computer Security Applications, pages 189–200. ACM, 2016. (Cité en page 39.)

[Banescu 2017] Sebastian Banescu, Christian Collberg et Alexander Pretschner. *Predicting the Resilience of Obfuscated Code Against Symbolic Execution Attacks via Machine Learning*. Dans 26th USENIX Security Symposium (USENIX Security 17), pages 661–678, Vancouver, BC, 2017. USENIX Association. (Cité en page 33.)

[Beddoe 2004a] Marshall Beddoe. *Network Protocol Analysis using Bioinformatics Algorithms*. `http://www.4tphi.net/~awalters/PI/pi.pdf`, 2004. (Cité en pages 11 et 12.)

[Beddoe 2004b] Marshall Beddoe. *Protocol Informatics Project*. `http://www.4tphi.net/~awalters/PI/PI.html`, 2004. (Cité en pages 4, 11, 12 et 15.)

[Bohlin 2008] Therese Bohlin et Bengt Jonsson. *Regular Inference for Communication Protocol Entities*. Technical Report 2008-024, Department of Information Technology, Uppsala University, Uppsala University, Sweden, 2008. (Cité en page 4.)

[Bossert 2011] G. Bossert, G. Hiet et T. Henin. *Modelling to Simulate Botnet Command and Control Protocols for the Evaluation of Network Intrusion Detection Systems*. Dans 2011 Conference on Network and Information Systems Security (SAR-SSI), pages 1–8, La Rochelle, France, 2011. IEEE. (Cité en pages 6, 12 et 14.)

[Bossert 2014a] Georges Bossert. *Exploiting Semantic for the Automatic Reverse Engineering of Communication Protocols.* phdthesis, Supélec, dec 2014. (Cité en pages 15 et 103.)

[Bossert 2014b] Georges Bossert, Frédéric Guihery et Guillaume Hiet. *Towards automated protocol reverse engineering using semantic information.* Dans Proceedings of the 9th ACM symposium on Information, computer and communications security, pages 51–62, Kyoto, Japan, jun 2014. ACM. (Cité en pages 12, 15 et 103.)

[Bridger 2015] Hahn Bridger, Nithyanand Rishab, Gill Phillipa et Johnson Rob. *Games Without Frontiers : Investigating Video Games as a Covert Channel.* Dans Proceedings of the 2016 IEEE European Symposium on Security and Privacy, IEEE European Symposium on Security and Privacy. IEEE, 2015. (Cité en page 43.)

[Caballero Bayerri 2010] Juan Caballero Bayerri. *Grammar and model extraction for security applications using dynamic program binary analysis.* PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 2010. (Cité en pages 6 et 22.)

[Caballero 2007a] Juan Caballero et Dawn Song. *Rosetta : Extracting Protocol Semantics using Binary Analysis with Applications to Protocol Replay and NAT Rewriting.* Technical Report CMU-CyLab-07-014, Carnegie Mellon University, Pittsburgh, USA, 2007. (Cité en pages 5, 12 et 20.)

[Caballero 2007b] Juan Caballero, Heng Yin, Zhenkai Liang et Dawn Song. *Polyglot : automatic extraction of protocol message format using dynamic binary analysis.* Dans Proceedings of the 14th ACM conference on Computer and communications security, CCS '07, pages 317–329, New York, NY, USA, 2007. ACM. (Cité en pages 4, 6, 11, 12, 19, 20 et 23.)

[Caballero 2009] Juan Caballero, Pongsin Poosankam, Christian Kreibich et Dawn Song. *Dispatcher : enabling active botnet infiltration using automatic protocol reverse-engineering.* Dans Proceedings of the 16th ACM conference on Computer and communications security, CCS '09, pages 621–634, New York, NY, USA, 2009. ACM. (Cité en pages 6, 12, 22 et 23.)

[Caballero 2012] Juan Caballero, Gustavo Grieco, Mark Marron, Zhiqiang Lin et David Urbina. *ARTISTE : Automatic Generation of Hybrid Data Structure Signatures from Binary Code Executions.* Technical Report TR-IMDEA-SW-2012-001, IMDEA Software Institute, Madrid, Spain, 2012. (Cité en pages 12 et 23.)

[Caballero 2013] Juan Caballero et Dawn Song. *Automatic protocol reverse-engineering : Message format extraction and field semantics inference.* Computer Networks, vol. 57, no. 2, pages 451–474, février 2013. (Cité en pages 12, 22 et 44.)

[Campana 2009a] Gabriel Campana. *Fuzzgrind : un outil de fuzzing automatique.* Dans Symposium sur la Sécurité des Technologies de l'Information et de la Communication, SSTIC, Rennes, France, 2009. SSTIC. (Cité en page 24.)

[Campana 2009b] Gabriel Campana. *Fuzzgrind : an automatic fuzzing tool*, 2009. (Cité en pages 12 et 24.)

[Cappaert 2010] Jan Cappaert et Bart Preneel. *A general model for hiding control flow.* Dans Proceedings of the tenth annual ACM workshop on Digital rights management, pages 35–42. ACM, 2010. (Cité en page 38.)

[Ceccato 2015] Mariano Ceccato, Andrea Capiluppi, Paolo Falcarin et Cornelia Boldyreff. *A large study on the effect of code obfuscation on the quality of java code.* Empirical Software Engineering, vol. 20, no. 6, pages 1486–1524, 2015. (Cité en page 33.)

[Cho 2001] W Cho, I Lee et S Park. *Against intelligent tampering : Software tamper resistance by extended control flow obfuscation.* Dans Proc. World Multiconference on Systems, Cybernetics, and Informatics, 2001. (Cité en page 38.)

[Cho 2010] Chia Yuan Cho, Domagoj Babić, Eui Chul Richard Shin et Dawn Song. *Inference and Analysis of Formal Models of Botnet Command and Control Protocols.* Dans Proceedings of the 17th ACM Conference on Computer and Communications Security, CCS '10, pages 426–439, New York, NY, USA, 2010. ACM. (Cité en pages 12 et 22.)

[Cho 2011] Chia Yuan Cho, Domagoj Babić, Pongsin Poosankam, Kevin Zhijie Chen, Edward XueJun Wu et Dawn Song. *MACE : model-inference-assisted concolic exploration for protocol and vulnerability discovery.* Dans Proceedings of the 20th USENIX conference on Security, SEC'11, page 19, Berkeley, CA, USA, aug 2011. USENIX Association. (Cité en pages 12 et 22.)

[Chow 2006] Jim Chow. *Understanding Data Lifetime.* PhD thesis, Stanford University, Stanford, CA, USA, 2006. AAI3219247. (Cité en page 22.)

[Collberg 1997] Christian Collberg, Clark Thomborson et Douglas Low. *A taxonomy of obfuscating transformations.* Rapport technique, Department of Computer Science, The University of Auckland, New Zealand, 1997. (Cité en pages 26, 32, 41, 51, 73 et 97.)

[Collberg 1998a] Christian Collberg, Clark Thomborson et Douglas Low. *Breaking abstractions and unstructuring data structures.* Dans Computer Languages, 1998. Proceedings. 1998 International Conference on, pages 28–38. IEEE, 1998. (Cité en pages 35 et 39.)

[Collberg 1998b] Christian Collberg, Clark Thomborson et Douglas Low. *Manufacturing cheap, resilient, and stealthy opaque constructs.* Dans Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pages 184–196. ACM, 1998. (Cité en page 36.)

[Collberg 2003] Christian Sven Collberg, Clark David Thomborson et Douglas Wai Kok Low. *Obfuscation techniques for enhancing software security*, décembre 23 2003. US Patent 6,668,325. (Cité en page 35.)

[Collberg 2007] Christian Collberg, Ginger Myles et Michael Stepp. *An empirical study of Java bytecode programs.* Software : Practice and Experience, vol. 37, no. 6, pages 581–641, 2007. (Cité en pages 35 et 38.)

[Comparetti 2009] P.M. Comparetti, G. Wondracek, C. Kruegel et E. Kirda. *Prospex : Protocol Specification Extraction.* Dans 2009 30th IEEE Symposium

on Security and Privacy, pages 110–125, Berkeley, USA, 2009. IEEE. (Cité en pages 12 et 20.)

[Cui 2006] Weidong Cui, Vern Paxson, Nicholas Weaver et Randy H. Katz. *Protocol-Independent Adaptive Replay of Application Dialog.* Dans Proceedings of the 13th Annual Network and Distributed System Security Symposium (NDSS), San Diego, USA, February 2006. Internet Society. (Cité en pages 4, 5, 12, 13 et 17.)

[Cui 2007a] Weidong Cui, Jayanthkumar Kannan et Helen J. Wang. *Discoverer : automatic protocol reverse engineering from network traces.* Dans Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium, SS'07, pages 14 :1–14 :14, Berkeley, CA, USA, 2007. USENIX Association. (Cité en pages 4, 12, 13, 15, 20 et 21.)

[Cui 2007b] Weidong Cui, M. Peinado, H.J. Wang et M.E. Locasto. *ShieldGen : Automatic Data Patch Generation for Unknown Vulnerabilities with Informed Probing.* Dans IEEE Symposium on Security and Privacy, 2007. SP '07, pages 252–266, Oakland, USA, 2007. IEEE. (Cité en page 21.)

[Cui 2008] Weidong Cui, Marcus Peinado, Karl Chen, Helen J. Wang et Luis Irun-Briz. *Tupni : automatic reverse engineering of input formats.* Dans Proceedings of the 15th ACM conference on Computer and communications security, CCS '08, pages 391–402, New York, NY, USA, 2008. ACM. (Cité en pages 6, 12, 21 et 23.)

[Dalla Preda 2007] Mila Dalla Preda. *Code obfuscation and malware detection by abstract interpretation.* PhD thesis, Universitad degli Studi di Verona, 2007. (Cité en pages 32, 33 et 39.)

[David 2016] Robin David, Sébastien Bardin, Thanh Dinh Ta, Laurent Mounier, Josselin Feist, Marie-Laure Potet et Jean-Yves Marion. *BINSEC/SE : A dynamic symbolic execution toolkit for binary-level analysis.* Dans 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), pages 653–656. IEEE, 2016. (Cité en page 33.)

[Duchêne 2016a] Julien Duchêne, Colas Le Guernic, Eric Alata, Vincent Nicomette et Mohamed Kaâniche. *Outils pour la rétro-conception de protocoles. Analyse et classification.* Techniques et sciences informatiques, vol. 35, pages 609–640, 12 2016. (Cité en page 3.)

[Duchêne 2016b] Julien Duchêne, Colas Le Guernic, Eric Alata, Vincent Nicomette et Mohamed Kaâniche. *Protocol reverse engineering : Challenges and obfuscation.* Dans 11th International Conference on Risks and Security of Internet and systems, Roscoff, France, septembre 2016. Springer. (Cité en page 76.)

[Duchêne 2017] Julien Duchêne, Colas Le Guernic, Eric Alata, Vincent Nicomette et Mohamed Kaâniche. *State of the art of network protocol reverse engineering tools.* Journal of Computer Virology and Hacking Techniques, pages 1–16, janvier 2017. (Cité en page 3.)

[Duchêne 2018] Julien Duchêne, Eric Alata, Vincent Nicomette, Mohamed Kaâniche et Colas Le Guernic. *Specification-Based Protocol Obfuscation.* Dans 2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN). IEEE, 2018. (Cité en page 76.)

[Dyer 2013] Kevin P. Dyer, Scott E. Coull, Thomas Ristenpart et Thomas Shrimpton. *Protocol Misidentification Made Easy with Format-transforming Encryption.* Dans Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security, CCS '13, pages 61–72, New York, NY, USA, 2013. ACM. (Cité en page 43.)

[Dyer 2015] Kevin P. Dyer, Scott E. Coull et Thomas Shrimpton. *Marionette : A Programmable Network Traffic Obfuscation System.* Dans Proceedings of the 24th USENIX Security Symposium, pages 367–382. USENIX Association, 2015. (Cité en pages 1, 41, 43 et 44.)

[Fielding 2014] Roy Fielding et Julian Reschke. *Hypertext transfer protocol (HTTP/1.1) : Message syntax and routing.* Rapport technique, Internet Engineering Task Force, 2014. (Cité en page 95.)

[Geddes 2013] John Geddes, Max Schuchard et Nicholas Hopper. *Cover Your ACKs : Pitfalls of Covert Channel Censorship Circumvention.* Dans Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security, CCS '13, pages 361–372, New York, NY, USA, 2013. ACM. (Cité en pages 42 et 43.)

[Google 2017] Google. *Protocol Buffers | Google Developers.* `https://developers.google.com/protocol-buffers/`, 2017. (Cité en pages 76 et 78.)

[Guihery 2012a] Frédéric Guihery et Georges Bossert. *The future of protocol reversing and simulation applied on ZeroAccess.* Dans 29C3 : 29th Chaos Communication Congress '12, Hambourg, Germany, décembre 2012. C-3. (Cité en pages 12 et 15.)

[Guihery 2012b] Frédéric Guihery et Georges Bossert. *Netzob : un outil pour la rétro-conception de protocoles de communication.* Dans Symposium sur la Sécurité des Technologies de l'Information et de la Communication, SSTIC, Rennes, France, 2012. SSTIC. (Cité en pages 6, 12, 15 et 56.)

[Hjelmvik 2010] Erik Hjelmvik et Wolfgang John. *Breaking and Improving Protocol Obfuscation.* Technical Report 2010-05, Chalmers University of Technology, Gothenburg, Sweden, 2010. (Cité en page 42.)

[Houmansadr 2013] Amir Houmansadr, Thomas J Riedl, Nikita Borisov et Andrew C Singer. *I want my voice to be heard : IP over Voice-over-IP for unobservable censorship circumvention.* Dans NDSS, 2013. (Cité en page 43.)

[Krueger 2010] Tammo Krueger, Nicole Krämer et Konrad Rieck. *ASAP : Automatic Semantics-Aware Analysis of Network Payloads.* Dans Christos Dimitrakakis, Aris Gkoulalas-Divanis, Aikaterini Mitrokotsa, Vassilios S. Verykios et Yücel Saygin, editeurs, Privacy and Security Issues in Data Mining and

Machine Learning, numéro 6549 de Lecture Notes in Computer Science, pages 50–63. Springer Berlin Heidelberg, Barcelona, Spain, 2010. DOI : 10.1007/978-3-642-19896-0_5. (Cité en pages 12 et 13.)

[Krueger 2012] Tammo Krueger, Hugo Gascon, Nicole Krämer et Konrad Rieck. *Learning Stateful Models for Network Honeypots.* Dans Proceedings of the 5th ACM Workshop on Security and Artificial Intelligence, AISec '12, pages 37–48, New York, NY, USA, 2012. ACM. (Cité en pages 12, 14 et 15.)

[Kuang 2018] Kaiyuan Kuang, Zhanyong Tang, Xiaoqing Gong, Dingyi Fang, Xiaojiang Chen et Zheng Wang. *Enhance virtual-machine-based code obfuscation security through dynamic bytecode scheduling.* Computers & Security, vol. 74, pages 202–220, 2018. (Cité en page 39.)

[Leita 2005] C. Leita, K. Mermoud et M. Dacier. *ScriptGen : an automated script generation tool for Honeyd.* Dans Computer Security Applications Conference, 21st Annual, pages 12 pp.–214, Tucson, USA, 2005. IEEE. (Cité en pages 5, 11, 12 et 15.)

[Leita 2008] Corrado Leita. *SGNET : automated protocol learning for the observation of malicious threats.* PhD thesis, Université de Nice, 2008. (Cité en page 11.)

[Li 2011] Xiangdong Li et Li Chen. *A Survey on Methods of Automatic Protocol Reverse Engineering.* Dans 2011 Seventh International Conference on Computational Intelligence and Security (CIS), pages 685–689, Hainan, China, 2011. IEEE. (Cité en page 3.)

[Li 2014] Shuai Li, Mike Schliep et Nick Hopper. *Facet : Streaming over videoconferencing for censorship circumvention.* Dans Proceedings of the 13th Workshop on Privacy in the Electronic Society, pages 163–172. ACM, 2014. (Cité en page 43.)

[Lim 2006] Junghee Lim, T. Reps et B. Liblit. *Extracting Output Formats from Executables.* Dans 13th Working Conference on Reverse Engineering, 2006. WCRE '06, pages 167–178, Benevento, Italy, 2006. IEEE. (Cité en pages 12, 17 et 23.)

[Lin 2008] Zhiqiang Lin, Xuxian Jiang, Dongyan Xu et Xiangyu Zhang. *Automatic Protocol Format Reverse Engineering through Context-Aware Monitored Execution.* Dans Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS), San Diego, USA, 2008. Internet Society. (Cité en pages 12, 20 et 22.)

[Lin 2009] Zhiqiang Lin, Ryan Riley et Dongyan Xu. *Polymorphing Software by Randomizing Data Structure Layout.* Dans DIMVA, volume 9, pages 107–126. Springer, 2009. (Cité en page 39.)

[Lin 2010] Zhiqiang Lin, Xiangyu Zhang et Dongyan Xu. *Automatic Reverse Engineering of Data Structures from Binary Execution.* Dans Proceedings of the 17th Annual Network and Distributed System Security Symposium (NDSS), San Diego, USA, 2010. Internet Society. (Cité en pages 12 et 23.)

[Lin 2011] Zhiqiang Lin. *Reverse Engineering of Data Structures from Binary*. PhD thesis, PURDUE UNIVERSITY, West Lafayette, Indiana, may 2011. (Cité en page 23.)

[Linn 2003] Cullen Linn et Saumya Debray. *Obfuscation of executable code to improve resistance to static disassembly*. Dans Proceedings of the 10th ACM conference on Computer and communications security, pages 290–299. ACM, 2003. (Cité en page 38.)

[Microsoft Corporation 2018] Microsoft Corporation. *Remote Procedure Call (Windows)*. `https://msdn.microsoft.com/en-us/library/windows/desktop/aa378651(v=vs.85).aspx`, 2018. (Cité en pages 76 et 79.)

[Microsoft 2017] Microsoft. *Informations de type au moment de l'exécution*. urlhttps ://msdn.microsoft.com/fr-fr/library/b2ay8610.aspx, 2017. (Cité en page 35.)

[Mohajeri Moghaddam 2012] Hooman Mohajeri Moghaddam, Baiyu Li, Mohammad Derakhshani et Ian Goldberg. *SkypeMorph : Protocol Obfuscation for Tor Bridges*. Dans Proceedings of the 2012 ACM Conference on Computer and Communications Security, CCS '12, pages 97–108, New York, NY, USA, 2012. ACM. (Cité en page 42.)

[Narayan 2015] John Narayan, Sandeep K Shukla et T Charles Clancy. *A Survey of Automatic Protocol Reverse Engineering Tools*. ACM Computing Surveys (CSUR), vol. 48, no. 3, page 40, 2015. (Cité en page 3.)

[Needleman 1970] Saul B. Needleman et Christian D. Wunsch. *A general method applicable to the search for similarities in the amino acid sequence of two proteins*. Journal of Molecular Biology, vol. 48, no. 3, pages 443–453, 1970. (Cité en page 11.)

[Nei 1983] Masatoshi Nei, Fumio Tajima et Yoshio Tateno. *Accuracy of estimated phylogenetic trees from molecular data*. Journal of Molecular Evolution, vol. 19, no. 2, pages 153–170, 1983. (Cité en page 11.)

[Nethercote 2007] Nicholas Nethercote et Julian Seward. *Valgrind : a framework for heavyweight dynamic binary instrumentation*. Dans Jeanne Ferrante et Kathryn S. McKinley, editeurs, Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007, pages 89–100. ACM, 2007. (Cité en page 24.)

[Newsome 2006] James Newsome, David Brumley, Jason Franklin et Dawn Song. *Replayer : automatic protocol replay by binary analysis*. Dans Proceedings of the 13th ACM conference on Computer and communications security, CCS '06, pages 311–321, New York, NY, USA, 2006. ACM. (Cité en pages 4, 5, 12, 17 et 20.)

[Reps 2005] Thomas Reps, Gogul Balakrishnan, Junghee Lim et Tim Teitelbaum. *A Next-Generation Platform for Analyzing Executables*. Dans 3rd Asian Sym-

posium on Programming Languages and Systems, pages 212–229. Springer-Verlag Berlin, Tsukuba, Japan, 2005. (Cité en page 17.)

[Roundy 2013] Kevin A Roundy et Barton P Miller. *Binary-code obfuscations in prevalent packer tools.* ACM Computing Surveys (CSUR), vol. 46, no. 1, page 4, 2013. (Cité en page 31.)

[Salwan 2018] Jonathan Salwan, Sébastien Bardin et Marie-Laure Potet. *Symbolic Deobfuscation : From Virtualized Code Back to the Original.* Dans International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment, pages 372–392. Springer, 2018. (Cité en page 39.)

[Samba Team 2018a] Samba Team. *How Samba was written.* `https://www.samba.org/ftp/tridge/misc/french_cafe.txt`, 2018. (Cité en page 5.)

[Samba Team 2018b] Samba Team. *Opening Windows to a Wider World.* `http://www.samba.org`, 2018. (Cité en page 5.)

[Schrittwieser 2011] Sebastian Schrittwieser et Stefan Katzenbeisser. *Code obfuscation against static and dynamic reverse engineering.* Dans Information Hiding, pages 270–284. Springer, 2011. (Cité en page 38.)

[Slowinska 2010] Asia Slowinska, Traian Stancescu et Herbert Bos. *Dynamic data structure excavation.* Technical Report IR-CS-55, Vrije Universiteit Amsterdam, Amsterdam, Danemark, 2010. (Cité en page 23.)

[Slowinska 2011] Asia Slowinska, Traian Stancescu et Herbert Bos. *Howard : A Dynamic Excavator for Reverse Engineering Data Structures.* Dans Proceedings of the 18th Annual Network and Distributed System Security Symposium (NDSS), San Diego, USA, 2011. Internet Society. (Cité en pages 12 et 23.)

[Swales 1999] Andy Swales. *Open modbus/tcp specification.* Technical Report, Schneider Electric, 1999. (Cité en page 47.)

[Tor team 2017] Tor team. *Obfsproxy.* `https://trac.torproject.org/projects/tor/wiki/doc/PluggableTransports/obfs4proxy`, 2017. (Cité en page 42.)

[Varda 2017] Kenton Varda. *Cap'n Proto.* `https://capnproto.org/`, 2017. (Cité en pages 76 et 78.)

[Wang 2000] Chenxi Wang, Jonathan Hill, John Knight et Jack Davidson. *Software tamper resistance : Obstructing static analysis of programs.* Rapport technique, Technical Report CS-2000-12, University of Virginia, 12 2000, 2000. (Cité en page 38.)

[Wang 2001] Chenxi Wang. *A security architecture for survivability mechanisms.* PhD thesis, University of Virginia, 2001. (Cité en pages 32 et 33.)

[Wang 2008] Rui Wang, XiaoFeng Wang, Kehuan Zhang et Zhuowei Li. *Towards Automatic Reverse Engineering of Software Security Configurations.* Dans Proceedings of the 15th ACM Conference on Computer and Communications

Security, CCS '08, pages 245–256, Limerick, Ireland, 2008. ACM. (Cité en pages 12 et 21.)

[Wang 2009] Zhi Wang, Xuxian Jiang, Weidong Cui, Xinyuan Wang et Mike Grace. *ReFormat : Automatic Reverse Engineering of Encrypted Messages*. Dans Michael Backes et Peng Ning, editeurs, Computer Security – ESORICS 2009, numéro 5789 de Lecture Notes in Computer Science, pages 200–215. Springer Berlin Heidelberg, Saint Malo, France, janvier 2009. (Cité en pages 12, 22 et 44.)

[Wang 2010] Yipeng Wang, Zhibin Zhang et Li Guo. *Inferring Protocol State Machine from Real-World Trace*. Dans Somesh Jha, Robin Sommer et Christian Kreibich, editeurs, Recent Advances in Intrusion Detection, numéro 6307 de Lecture Notes in Computer Science, pages 498–499. Springer Berlin Heidelberg, Ottawa, Canada, septembre 2010. DOI : 10.1007/978-3-642-15512-3_32. (Cité en pages 12 et 13.)

[Wang 2011] Yipeng Wang, Zhibin Zhang, Danfeng (Daphne) Yao, Buyun Qu et Li Guo. *Inferring Protocol State Machine from Network Traces : A Probabilistic Approach*. Dans Javier Lopez et Gene Tsudik, editeurs, Applied Cryptography and Network Security, numéro 6715 de Lecture Notes in Computer Science, pages 1–18. Springer Berlin Heidelberg, Nerja, Spain, jan 2011. (Cité en pages 12, 13 et 64.)

[Wang 2012] Qiyan Wang, Xun Gong, Giang TK Nguyen, Amir Houmansadr et Nikita Borisov. *Censorspoofer : asymmetric communication using ip spoofing for censorship-resistant web browsing*. Dans Proceedings of the 2012 ACM conference on Computer and communications security, pages 121–132. ACM, 2012. (Cité en page 43.)

[Wang 2015] Liang Wang, Kevin P. Dyer, Aditya Akella, Thomas Ristenpart et Thomas Shrimpton. *Seeing Through Network-Protocol Obfuscation*. Dans Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security, CCS '15, pages 57–69, New York, NY, USA, 2015. ACM. (Cité en page 44.)

[Weinberg 2012] Zachary Weinberg, Jeffrey Wang, Vinod Yegneswaran, Linda Briesemeister, Steven Cheung, Frank Wang et Dan Boneh. *StegoTorus : a camouflage proxy for the Tor anonymity system*. Dans Proceedings of the 2012 ACM conference on Computer and communications security, pages 109–120. ACM, 2012. (Cité en page 42.)

[Winter 2013] Philipp Winter, Tobias Pulls et Juergen Fuss. *ScrambleSuit : A Polymorphic Network Protocol to Circumvent Censorship*. Dans Proceedings of the 12th ACM Workshop on Privacy in the Electronic Society, WPES '13, pages 213–224, New York, NY, USA, 2013. ACM. (Cité en page 42.)

[Winter 2017a] Philipp Winter. *Obfs2*. `https://github.com/NullHypothesis/obfsproxy/blob/master/doc/obfs2/obfs2-threat-model.txt`, 2017. (Cité en page 42.)

[Winter 2017b] Philipp Winter. *Obfs3*. `https://github.com/NullHypothesis/obfsproxy/blob/master/doc/obfs3/obfs3-threat-model.txt`, 2017. (Cité en page 42.)

[Wondracek 2008] Gilbert Wondracek, Paolo Milani Comparetti, Christopher Krügel et Engin Kirda. *Automatic network protocol analysis*. Dans Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS), San Diego, USA, 2008. Internet Society. (Cité en pages 12 et 20.)

[Wroblewski 2002] Gregory Wroblewski. *General Method of Program Code Obfuscation*. PhD thesis, Institute of Engineering Cybernetics, Wroclaw University of Technology, 2002. (Cité en page 38.)

[Zalewski 2017] Michał Zalewski. *American Fuzzy Lop*. `http://lcamtuf.coredump.cx/afl/technical_details.txt`, 2017. (Cité en pages 12 et 24.)

[Zeng 2015] Junyuan Zeng et Zhiqiang Lin. *Towards Automatic Inference of Kernel Object Semantics from Binary Code*. Dans 18th International Symposium, RAID 2015, volume 9404, pages 538–561, Kyoto, Japan, novembre 2015. Springer. (Cité en pages 12 et 23.)

Abstract

There are more and more protocols. Many of them have their specification available for interoperability purpose for instance. However, when it comes to intellectual property, this specification is kept secret. Attackers might use a wrongly implemented protocol to compromise a system, if he has access to the specification, it's attack would be far more efficient. Even if he does not have access to the specification, he can reverse-engine it. Thus, create protocols hard to reverse is interesting. In this thesis, we develop a novel approach of protocol protections to make protocol reverse engineering more complex. We apply some obfuscations on protocol message format, furthermore, we do it automatically from the original protocol specification.

Firstly, we have analyzed more than 30 different contributions of protocol reverse engineering tools. We retrieved the following elements : 1) Protocol reverse engineering tools try to infer regular models ; 2) They suppose that the parsing is done from left to right ; 3) They delimit fields based on well-known delimiters or with ad-hoc techniques ; 4) They cluster messages based on pattern similarity measures. Thus, to make protocol reverse harder, one can create protocols which does not respect theses statements.

Secondly, we have proposed a model of message format on which obfuscations can be applied. With this model, we also provide some atomic obfuscations which can be composed. Each obfuscation target one or more protocol reverse engineering hypothesis. Obfuscation composition ensures effectiveness of our solution and makes protocol reverse-engineering more complex. This model is used to automatically generate code for parser, serializer and accessors. This solution is implemented into a framework we called *ProtoObf*. *ProtoObf* is used to evaluate obfuscations performance. Results show an increase of protocol complexity with the number of obfuscation composition while costs (particularly the serialized buffer size) stay low.

Keywords : Security, Protocols, Reverse-Engineering, Obfuscations, Protocol specification

---

Résumé

Il existe de plus en plus de protocoles de communications différents. La spécification de beaucoup d'entre eux est disponible. Cependant, quand il s'agit de moyens de communication propriétaires, cette spécification est gardée secrète : un attaquant qui aurait accès à cette spécification pourrait compromettre un système utilisant ce protocole. Même s'il n'a pas accès à cette spécification, l'attaquant peut l'obtenir par rétro-conception. Ainsi, il est intéressant de créer des protocoles qui sont difficiles à rétro-concevoir. Dans cette thèse, nous proposons une nouvelle approche spécifiquement développée pour rendre complexe la rétro-conception de protocole. Nous appliquons pour cela des obfuscations au format du message et ceci de façon automatique à partir de la spécification du protocole.

Pour cela, nous avons dans un premier temps étudié plus de 30 contributions différentes concernant des outils de rétro-conception de protocole et en avons tiré des conclusions suivantes : 1) les outils de rétro-conception de protocole pratiquent l'inférence de modèles réguliers ; 2) ils supposent que le parsing d'un message s'effectue de gauche à droite ; 3) ils délimitent le message en champs d'après des délimiteurs bien connus ou via des algorithmes ad-hoc ; 4) ils regroupent les messages d'après des mesures de similarité sur des patterns. Ainsi, pour créer un protocole difficile à rétro-concevoir, une solution est de s'assurer que le protocole ne respecte pas ces conditions.

Dans un second temps, nous avons donc proposé un modèle de format de messages qui permet l'application d'obfuscations. Nous avons défini des obfuscations atomiques qui peuvent être composées. Chacune de ces obfuscations cible une ou plusieurs des hypothèses des outils de rétro-conception. La composition des obfuscations assure l'efficacité de notre solution et rend la rétro-conception de protocole complexe. Ce modèle est utilisé pour générer automatiquement le code du parseur, du sérialiseur et des accesseurs. Cette solution est implémentée dans un prototype nommé *ProtoObf* grâce auquel nous avons pu évalué les performances des obfuscations. Les résultats montrent une nette augmentation de la complexité de la rétro-conception avec le nombre de compositions d'obfuscation tandis que les coûts induits (particulièrement la taille du buffer sérialisé) restent bas.

Mots clés : Sécurité, Protocoles, Rétro-Conception, Obfuscations, Spécification de protocoles

---