Exploiting Semantic for the Automatic Reverse Engineering of Communication Protocols.

Georges Bossert

Supélec

Nº d'ordre : 2014-27-TH

SUPELEC

École Doctorale MATISSE

"Mathématiques, Télécommunications, Informatique, Signal, Systèmes Électroniques"

THÈSE DE DOCTORAT

DOMAINE : STIC

Spécialité : Informatique

Soutenue le **17 décembre 2014**

par :

Georges BOSSERT

Exploiting Semantic for the Automatic Reverse Engineering of Communication Protocols

Directeur de thèse :	Ludovic MÉ	Professeur à Supélec
Composition du jury :		
Président du jury :	François BODIN	Professeur à l'Université de Rennes 1
Rapporteurs :	Colin DE LA HIGUERA	Professeur à l'Université de Nantes
	Christopher KRUEGEL	Professeur à l'Université de Californie SB
Examinateurs :	Hervé DEBAR	Professeur à Télécom SudParis
	Benjamin MORIN	Chef de Laboratoire, ANSSI
Membre invité :	Dominique CHAUVEAU	Chef du département IMPS, DGA MI
Encadrants :	Frédéric GUIHÉRY	Responsable R&D, AMOSSYS
	Guillaume HIET	Professeur assistant à Supélec

Remerciements

À l'issue de ces quatre années, je suis persuadé que la thèse n'est pas un travail solitaire. Elle est le résultat d'un effort conjoint entre le doctorant et son encadrement. L'environnement humain au sein duquel évoluent ces acteurs participe également pleinement aux résultats de ces travaux. Il m'apparaît donc primordial de remercier ces personnes dont la générosité, la bienveillance et la bonne humeur m'ont apporté la force de progresser et de terminer cette thèse.

En premier lieu, je souhaite remercier mon encadrant Guillaume HIET, sans qui ces travaux n'auraient jamais abouti. L'énergie, la rigueur et la patience qu'il a su déployer pour m'encadrer se retrouvent distillées tout au long de ce manuscrit. Son expertise et son expérience furent également exploitées à maintes reprises pendant l'élaboration des contributions exposées. De la même manière, l'encadrement assuré par Fredéric GUIHÉRY fut précieux. L'intérêt qu'il a su porter à ce travail a largement dépassé le cadre d'un encadrement et ses contributions actives au projet NETZOB se retrouvent donc tout au long de ce manuscrit.

Je souhaite également exprimer toute ma gratitude à la société AMOSSYS et ses directeurs Frédéric RÉMI et Christophe DUPAS pour leur confiance. La bienveillance et l'intérêt que les équipes d'AMOSSYS ont porté à mes travaux ont très largement participé aux résultats de cette thèse. Nos interminables débats techniques (ou non) et les discussions avec les membres du conseil stratégique du jeudi soir furent de véritables bouffées d'air frais, je les remercie tous très sincèrement pour ça.

Ces remerciements seraient incomplets si je n'en adressais pas à Olivier TÉTARD qui s'est beaucoup investi pour créer un environnement humain et technique pérein autour du projet NETZOB. Ses conseils m'ont accompagné tout au long de cette thèse, je le remercie très sincèrement pour ça. Je souhaite également associer à cette thèse l'ensemble des contributeurs au projet NETZOB qui ont su donner vie à un projet unique et plein de promesses.

Mes remerciements vont également aux membres de l'équipe CIDre de SUPELEC pour m'avoir accueilli et pour l'ambiance de travail très agréable qu'ils ont su créer. Ludovic MÉ en tant que directeur de l'équipe et de ma thèse a joué un rôle très important pendant ces quatre années, je le remercie très chaleureusement.

De très nombreuses personnes m'ont accompagné pendant ces années, je ne peux malheureusement pas les citer pour cette raison. La famille et les amis furent des points de repère vitaux. Leurs encouragements et la chaleur qu'ils m'ont prodigué ont très largement produit le soutien affectif nécessaire à ce travail doctoral. Merci à eux.

Enfin, j'exprime ma gratitude aux membres du jury qui se sont rendus disponibles pour la

soutenance et pour les conseils prodigués sans lesquels ces travaux seraient certainement moins riches. Je suis particulièrement reconnaissant à Colin DE LA HIGUERA et Christopher KRUEGEL de l'intérêt qu'ils ont manifesté à l'égard de cette recherche en s'engageant en tant que rapporteurs.

Résumé en français

Introduction

Les protocoles de communication sont fondamentaux pour la communication des différents composants d'un système d'information. Ils spécifient les règles à suivre pour assurer la transmission de données inhérentes à toute communication. Avec l'accroissement des besoins en interconnexion des systèmes informatiques, l'emploi de protocoles de communication tend à se généraliser au sein des systèmes d'informations personnels, industriels et militaires. Malheureusement, il est également bien connu que les protocoles de communication peuvent être vulnérables à des attaques [48, 126, 118, 128]. Ces vulnérabilités peuvent être exploitées pour mettre à mal l'intégrité, la disponibilité et/ou la confidentialité des données et des applications. Certains protocoles, comme ceux utilisés par les réseaux de zombies [129, 136, 6], ont même été créés dans le seul but de réaliser des attaques informatiques.

Pour remédier aux vulnérabilités d'un protocole, plusieurs solutions existent telles que l'évaluation de sécurité des implémentations ou l'emploi de produits de sécurité dédiés tels que des systèmes de détection d'intrusion réseau (NIDS) ou des parefeu applicatifs. Cependant, la qualité des résultats apportés par ces solutions dépend principalement de la connaissance des spécifications des protocoles. Il est aisé d'obtenir cette connaissance si les spécifications du protocole sont disponibles. Cependant, si le protocole est non-documenté et/ou propriétaire, un expert doit d'abord rétro-concevoir les spécifications du protocole. Étant donné la complexité de certains protocoles, cette opération peut être très coûteuse lorsqu'elle est exécutée manuellement.

Objectifs de nos travaux

Cette thèse expose des contributions pour automatiser et améliorer les opérations de rétro-conception d'un protocole de communication. Plus précisément, ces contributions visent à obtenir un modèle précis des spécifications d'un protocole inconnu tout en réduisant le temps de calcul nécessaire et en augmentant la furtivité de l'apprentissage par rapport aux solutions précédentes. Ce travail embrasse la nécessité d'une méthode efficace et rapide pour la rétro-conception d'un protocole afin d'aider les auditeurs de sécurité, les évaluateurs de sécurité et les développeurs de produits de sécurité dans leur travail contre les cyber-attaques.

Un protocole de communication peut se définir comme un ensemble de règles qui régissent la nature de la communication, des données échangées et des comportements dépendant de l'état des systèmes qui participent à la communication [65]. Ces règles définissent le vocabulaire et la grammaire du protocole et dans une certaine mesure peuvent être considérées comme similaires aux règles qui régissent les langages de programmation. Le vocabulaire d'un protocole établit l'ensemble des messages valides et leurs formats, tandis que la grammaire du protocole spécifie l'ensemble des échanges valides de messages. Ces règles peuvent être rendues complexes par le besoin d'assurer des communications entre des systèmes évoluant dans des milieux très diversifiés.

La rétro-conception de protocoles de communication désigne les procédés utilisés pour obtenir les spécifications d'un protocole non-documenté. Cette connaissance des spécifications du protocole est très précieuse pour mener à bien de nombreuses opérations ayant trait à la sécurité. Par exemple, les audits de sécurité des systèmes de contrôle industriels impliquent souvent l'analyse de matériels et de logiciels propriétaires. En outre, l'audit d'un tel système propriétaire exige d'acquérir des connaissances suffisantes sur les fonctionnalités offertes par son protocole. Si aucune documentation n'est disponible, comme lors d'un audit en boîte noire, l'expert est obligé de rétro-concevoir le protocole. L'importance de l'automatisation de ce processus est bien établie compte tenu de la difficulté et le temps qu'il nécessite lorsqu'il est exécuté manuellement.

En plus des évaluations et des audits de sécurité, la rétro-conception d'un protocole est également utile pour les développeurs et les évaluateurs de produits de sécurité tels que les systèmes de détection d'intrusion, les parefeu applicatifs, les outils de supervision de réseau et pots de miel. Au final, trois principales motivations nous ont conduit à mener ces travaux : 1) le besoin en rétro-conception de protocoles pour les évaluations de sécurité, 2) la création de règles de détection d'intrusion réseau à partir des résultats de la rétro-conception d'un protocole inconnu et 3) l'exploitation de la rétro-conception des protocoles pour générer un trafic réaliste et contrôlable de réseaux de zombies.

En outre, les travaux existants dans le domaine de l'automatisation de la rétro-conception d'un protocole tendent soit à inférer des spécifications incomplètes soit à nécessiter trop de stimulation de l'implémentation du protocole, avec le risque d'être vaincu par des techniques de contre-inférence. En outre, aucun de ces travaux ne permettent d'obtenir des spécifications suffisamment détaillées pour permettre à terme la simulation du protocole inféré. Enfin, le temps de calcul requis par les précédents travaux peut être important, ce qui empêche leurs utilisations lorsque que le domaine d'emploi nécessite une grande réactivité. Les objectifs de cette thèse sont donc les suivants :

Objectif 1 La solution proposée doit permettre d'obtenir un modèle juste, correct et précis des spécifications d'un protocole non-documenté. Ce modèle doit couvrir le vocabulaire et la grammaire du protocole.

Objectif 2 La solution de rétro-conception doit être plus rapide que les travaux existants pour inférer le modèle d'un protocole.

Objectif 3 Nos travaux doivent permettre d'augmenter la furtivité de l'inférence d'un protocole par rapport aux autres travaux.

Ces travaux portent sur les deux principaux aspects de la rétro-conception d'un protocole, à savoir : l'inférence de sa définition syntaxique (le vocabulaire du protocole) et de sa définition grammaticale (la grammaire du protocole). Nos travaux se distinguent de l'état de l'art de pars les algorithmes que nous proposons. Ces algorithmes exploitent notamment des informations sémantiques pour inférer le vocabulaire et la grammaire d'un protocole. Les contributions présentées dans cette thèse sont organisées en deux grandes parties. La première partie du manuscrit détaille l'approche retenue pour inférer le vocabulaire d'un protocole, la seconde partie traite de l'inférence de la grammaire.

Contributions pour la rétro-conception du vocabulaire

La méthodologie proposée dans la première partie de cette thèse s'articule autour de l'emploi de la sémantique pour l'amélioration de l'inférence du format des messages.

Plus précisément, ces travaux exploitent la présence d'information contextuelle dans les messages pour identifier les messages équivalents d'un point de vue protocolaire et pour inférer leurs structures en champs. À cette fin, nous proposons une extension de l'algorithme de Needleman&Wunsch [97] permettant d'introduire des contraintes sémantiques lors de l'alignement de messages.

Nous proposons également de prendre en compte les actions réalisées par l'implémentation du protocole pendant la capture des messages pour améliorer leur classification. Ces opérations sont réalisées au travers de plusieurs étapes de classification et de pré-classification. De cette manière, nous favorisons l'identification de messages équivalents avant d'inférer leur structure en champs. Cette approche permet également d'optimiser le temps de calcul. En outre, nous proposons également une solution efficace et rapide pour la découverte de relations entre les champs d'un ou de plusieurs messages (champs taille, CRC, ...). Cette solution repose sur un algorithme de corrélation, ce qui permet de réduire le temps de calcul tout en supportant d'éventuelles erreurs de classification.

Au terme de cette partie, une comparaison expérimentale des différentes approches existantes est présentée. Cette étude permet de mettre en évidence les avantages et inconvénients des différentes solutions pour la rétro-conception du vocabulaire d'un protocole. A notre connaissance, aucun des travaux précédents ne s'était attaché à réaliser une telle étude. Les résultats obtenus justifient également de l'intérêt de notre approche : les spécifications obtenues avec notre approche sont plus correctes, plus concises et plus précises.

Contributions pour la rétro-conception de la grammaire

La seconde partie de ce manuscrit détaille nos contributions pour l'inférence automatisée de la grammaire d'un protocole. Comme indiqué par G. Holzman, la grammaire d'un protocole de communication représente les séquences valides de messages reçus et émis. La théorie des machines à états étant étroitement liée à la théorie des langages formels, l'emploi d'automates est adapté à la modélisation des règles qui établissent ces séquences. Parmi tous les modèles d'automates existants, nous avons retenu une *Machine à états finis (FSM)* disposant de sorties, aussi appelée *machine de Mealy*.

L'algorithme de référence pour l'inférence active et automatisée d'une machine à états est l'algorithme LSTAR, proposée par D. Angluin. Cet algorithme permet d'inférer une machine de Mealy décrivant l'ensemble des séquences de messages acceptées par le protocole cible. Cependant, comme détaillé dans cette thèse, le nombre requis de requêtes, le temps de l'inférence et la non-furtivité de cet algorithme peuvent empêcher son exploitation sur des protocoles de communication complexes, *i.e.* disposant d'un automate à nombreux états. Pour répondre à cette problématique, nous proposons un algorithme d'inférence de type ≪ diviser pour régner ≫. L'objectif de cette approche est de limiter la complexité intrinsèque de l'algorithme LSTAR afin de réduire le nombre de requêtes nécessaires, réduire le temps d'inférence et augmenter la furtivité du procédé.

Notre approche repose sur l'hypothèse que la grammaire d'un protocole peut être décomposée en plusieurs éléments plus simples, que nous appelons sous-grammaires. Comme indiqué par H. Zafar [61], la décomposition d'automates complexes en éléments plus simples a fait l'objet de nombreux ouvrages [11, 45, 60, 10]. Dans notre travail, nous cherchons à tirer parti de ce concept de décomposition de machines à états pour optimiser la rétro-conception de la grammaire d'un protocole de communication. Plus précisément, notre solution repose sur l'observation que l'implémentation d'un protocole expose différentes actions à son utilisateur. Toutes ces actions participent à l'objectif général du protocole tels que l'authentification du client ou la création d'un répertoire dans le protocole FTP. Une action peut être considérée comme une composante fonctionnelle du protocole et désigne un sous-ensemble du vocabulaire de protocole et de sa grammaire.

L'algorithme proposé réalise donc l'inférence de chaque sous-grammaire du protocole de manière indépendante. Pour cela, une première étape d'inférence est menée afin d'obtenir le vocabulaire associé à chaque action. Une fois ce vocabulaire obtenu, une instance de l'algorithme LSTAR est exécutée pour chaque action à inférer. Les grammaires obtenues sont ensuite fusionnées pour obtenir la grammaire du protocole.

Pour évaluer l'intérêt de notre approche, une étude comparative expérimentale est proposée. Celle-ci a consisté à comparer les résultats obtenus avec l'algorithme LSTAR et avec notre approche. Comme expliqué, les résultats démontrent que la décomposition de la grammaire cible permet effectivement de réduire considérablement le temps d'exécution ainsi que le nombre de requêtes envoyées à l'implémentation et augmente la furtivité de l'inférence.

Conclusion

Les résultats obtenus par notre solution d'inférence de vocabulaire renvoient de meilleurs résultats que les travaux existants quant à la précision du modèle, son exhaustivité et son exactitude (**objectif 1**). En outre, nous pensons que l'utilisation d'un algorithme de classification multi-étapes et d'une solution d'identification de la relation basée sur une mesure de corrélation permet de réduire la complexité globale de l'inférence et ainsi de limiter le temps de calcul (**objectif 2**). Pour finir, notre solution d'inférence du vocabulaire d'un protocole repose sur une approche passive assurant sa furtivité (**objectif 3**).

En ce qui concerne notre solution d'inférence grammaticale, nous affirmons que les objectifs de cette thèse sont partiellement remplis. En effet, la technique que nous proposons s'appuie sur des heuristiques qui peuvent conduire à des résultats incorrects et/ou incomplets sur certains protocoles. Néanmoins, nous expliquons également dans cette thèse que cette problématique peut être résolue en introduisant une politique par défaut pour modéliser les transitions non apprises. En outre, l'étude comparative confirme également que notre approche peut être utilisée pour déduire une grammaire précise, correcte et presque complète d'un protocole inconnu (**objectif 1**). En outre, les résultats exposés montrent que notre solution nécessite moins de temps de calcul (**objectif 2**) tout en étant plus furtive (**objectif 3**) que l'état de l'art.

Pour conclure, cette thèse a également donné lieu à la réalisation d'un outil open-source, appelé Netzob [1], qui met en œuvre nos solutions pour aider les experts en sécurité dans leurs tâches de rétro-conception d'un protocole. Il s'agit actuellement, à notre connaissance, de l'outil publiquement disponible le plus avancé pour la rétro-conception semi-automatique de protocoles.

1. Netzob : `http://www.netzob.org`

Abstract

Network security products, such as NIDS or application firewalls, tend to focus on application level communication flows to perform their analysis. However, adding support for new proprietary and often undocumented protocols, implies the reverse engineering of these protocols. Currently, this task is performed manually. Considering the difficulty and time needed for manual reverse engineering of protocols, one can easily understand the importance of automating this task. This is even given more significance in today's cybersecurity context where reaction time and automated adaptation become a priority.

Current work in the field of automated protocol reverse engineering either infer incomplete protocol specifications or require too many stimulation of the targeted implementation with the risk of being defeated by counter-inference techniques. Besides, none of these work infer detailed enough specification that could support the simulation of the reversed protocol. In addition, the computation time required by previous reverse engineering work can be enormous which can prevent their uses where high responsiveness is mandatory. Finally, too few previous works have resulted in the publication of tools that would allow the scientific community to experimentally validate and compare the different approaches.

This thesis exposes a practical approach for the automatic reverse engineering of undocumented communication protocols. This work leverages the semantic of the protocol to improve the quality, the speed and the stealthiness of the inference process when applied on complex protocols. Our work covers the two main aspects of the protocol RE, the inference of its syntactical definition (the protocol vocabulary) and of its grammatical definition (the protocol grammar). The algorithms we propose uses the semantic definition in both domains. We conducted multiple experiments to validate our approach by comparing previous state-of-the-art work against our algorithms. We also propose an open-source tool, called Netzob, that implements our work to help security experts in their protocol reverse engineering tasks. We claim Netzob is the most advanced published tool that tackles issues related to the reverse engineering and the simulation of undocumented protocols.

Contents

Chapter 1

Introduction

Communication protocols play a major role as a fundamental necessity that enables communication between the different components of computer systems. As those systems become more and more connected [75, 120], communication protocols are frequently used at different levels of computer systems. Sadly, it is also well-known that communication protocols can be vulnerable to attacks [48, 126, 118, 128]. Attackers can crash or hijack victims by sending unexpected or malformed messages that exploit bugs or inadequate defenses in protocol implementations. Some protocols, such those used by Botnets [129, 136, 6] were even created for the sole purpose of computer attacks. To address protocol vulnerabilities, many solutions exist. Among them, the security evaluation of communication protocol implementations is often considered. Another solution is to rely on various security products such as Intrusion Detection System (IDS) that can detect attacks on protocol implementations. However, the quality of a product that analysis communication for security flaws mostly depend on its knowledge over the protocol specifications. It is straightforward to obtain this knowledge if the protocol specifications are available. Conversely, if the protocol is undocumented and/or proprietary, the expert must reverse the protocol implementation to obtain its specifications. In regards to the complexity of some existing protocols, this operation can be very expensive when executed manually. For this reason, this thesis exposes our solutions to reverse engineer a communication protocol with the key objectives of obtaining a fine-grained model of the protocol specification while reducing the required computation time and increasing its stealthiness in comparison to previous works. Indeed, our work embraces the need of an efficient and automated protocol reverse engineering technique that helps security auditors, security evaluators and security product developers in their work against cyber attacks.

In the remainder, Section 1.1 gives some insights over the definition of a communication protocol and illustrates the omnipresence of undocumented ones. We then discuss the reasons that prompted us to embrace the field of automated protocol reverse engineering in Section 1.2. Consequently, Section 1.3 summarizes our problem statement, our objectives and the contributions that are exposed all along this thesis. Finally, we present in Section 1.4 our dissertation outlines.

1.1 Insights on Communication Protocols

This Section gives some insights on communication protocols. The remainder is organized as follows: we expose a basic definition of a communication protocol in Section 1.1.1 and then show the omnipresence of protocols in Section 1.1.2 by means of three examples: Internet and LAN protocols, industrial protocols and malware protocols.

1.1.1 Basic Definition of a Communication Protocol

A communication protocol can be defined as a set of rules that govern the nature of the communication, the exchanged data and any state-dependent behaviors that participate in the communication [65]. These rules define the vocabulary and the grammar of the protocol and to some extend can be seen as similar to the rules that govern programming languages. The protocol vocabulary defines the set of valid messages and their format, while the protocol grammar specifies the set of valid protocol exchanges. These rules can be complex since they can be designed to ensure the protocol usages in very diverse settings. The unreliability of transmission links is an example of a recurrent issue these rules must address to fulfill even the most basic requirements. To ease their design, a layered architecture of protocols is promoted by the Open Systems Interconnection (OSI) model [71] and retained in very most network related communication protocols. This model decomposes complex protocol into simpler "single task", cooperating protocols. With this conceptual model, called **protocol layering**, a protocol covers its specific layer functions, relies on sub layers functions and provides its features to upper-layer protocols. For example, the Transmission Control Protocol (TCP) [49] is a transport protocol, *i.e.* member of the fourth layer named the transport layer, that ensures data transfer reliability through error control, flow control and data segmentation features. With this layer approach, the developer of a new protocol can base its work over the TCP reliability functions and so focus on its specific aspects usually implemented at the application layer. The OSI model defines seven layers covering a large spectrum of typical communication functions from its electrical aspects with the physical layer, up to the end-user with the application layer. These layers are illustrated in Figure 1.1. When an application sends a message to a remote application, this message successively goes through all the layers from the application one to the physical one.

From a practical point view, protocol layering approach impacts the format and the content of emitted messages. Indeed, a network communication involves a set of protocols, one protocol for each layer. Thus, a message is successively handled by protocols that belong to underlying layers. To ensure their roles, each protocol can optionally prepend and/or append control information to the message it received from its upper layer protocol. In the ISO nomenclature, the term of Protocol Data Unit (PDU) denotes a message extended with control information. Indeed, the term of (N)-PDU denotes a message that belongs to the N-*th* layer. Control information often contains parameters such as source and destination identifiers, data lengths and timestamps. Besides, the layered architecture of protocols often implies that two communicating systems often use multiple protocols to handle their exchanges, usually, one protocol per layer.

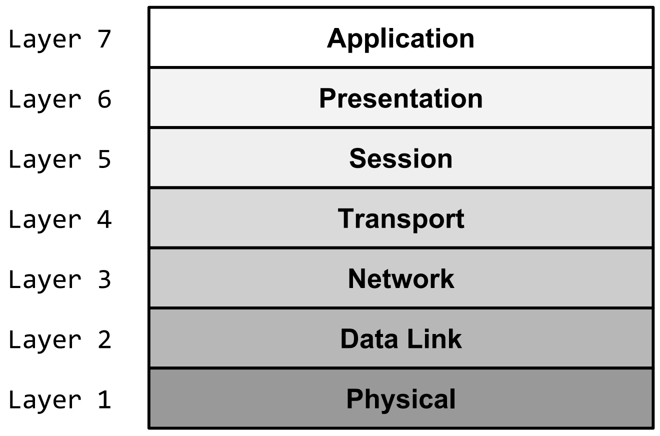


Figure 1.1 – Layering architecture of the OSI model.

1.1.2 Communication Protocols are Everywhere

A communication protocol is mandatory to ensure a transmission of information. Every time two or more systems communicate, at least one communication protocol is used. It exists a very large panel of tools and systems that need to communicate. In the following, we point out the fact that protocols are used in various heterogeneous application domains. We cannot provide an exhaustive list of all the existing protocols as, to the best of our knowledge, no such list exists mostly due to the high rate at which protocols are created. To illustrate that protocols can be found everywhere, we retained three common application domains (Internet, industrial sector and malware) and detail their usage of communication protocols.

Typical Internet and LAN Usages of Communication Protocols

Internet is the biggest network of networks that exists. Some of these networks are Local Area Network (LAN) as they interconnect computers within a limited area such as a home. Such network relies on various distributed services and so protocols. Some of them are fundamental for the stability of this network as with the Transmission Control Protocol (TCP) [109] protocol that ensures the reliability of message exchanges. Naturally, Internet architecture relies on the Internet Protocol (IP) [108] protocol that provides the required routing features to support message exchanges across Internet actors. Besides, such network make an heavy use of protocols that rely on this TCP/IP layer. Among them, the DNS protocol provides the domain name resolution system to map an IP address to a more human-readable domain name. Obviously, the HTTP protocol that enables data exchanges such as during a web navigation, is one of most frequent protocol used on these networks. Their specifications are freely available to ensure their adoptions.

Multitudes of other protocols are also used on the Internet such as peer-to-peer protocols (*e.g.* Napster, Gnutella, FastTrack, Bittorrent) and instant messaging protocols (*e.g.* XMPP, MSNP, Skype). Internet also hosts various data exchanges such as timing information (*e.g.* NTP, PTP, TPSN) and file transfers (*e.g.* FTP, RSYNC, RCP). Private companies along with public sectors created numerous communication protocols to address their needs and those of their users. The specification of some of these protocols are not available. The Skype protocol [1] is a famous example of such proprietary protocol. They are multiple reasons that can explain that an organization does not release its protocol specification. One reason can be that the time required to write the specification in a proper way can be seen as a break in an innovation race. Another common reason, is that communication protocols are often expensive to develop. By publishing their specification, an organization releases some parts of its intellectual propriety that may latter profit to its competitors.

Industrial Usages of Communication Protocols

Last years have seen an increase of communicating industries. Also known as Smart Industries, these industries have a high degree of flexibility in production by means of a network-centric approach. The term of Smart Grid is an illustration of this tendency applied to the industrial domain of electricity production. Driven by latest ICT technologies, these industries make an heavy use of standardized or proprietary communication protocols to interconnect all their equipment. However, industrial specific needs in terms of high reliability spawned the creation of specific protocols. Besides, the complexity induced by industrial automation systems brought the classification of industrial networks and of their protocols in several different categories [46]. Each category denotes an appropriate communication level, which places different requirements on the communication network. Sensors, actuators and device buses constitutes the first category of industrial networks called *field-level networks*. Control buses are organized in *control network* while the top level industrial systems form the *information network* and gather information produced by lowest level systems to manage the whole automation system. Multiples of communication protocols were created to address the needs of each levels. For example ProfiBus [2], DeviceNet [100] or Bitbus [3] are protocols that participate in field-level networks with more than 70 other *major* protocols [121]. The largest listing of industrial protocols we have found, reveals that more than 180 common automation protocols exist [4]. Most of these protocols are developed for a local usage that often support no interconnection. Such protocol are often not conceived for their adoption across heterogeneous networks. For this reason, protocol creators do not require the adoption of their protocols by multiple actors which explains that numerous industrial protocols are proprietary and their specifications not available.

Malware Usages of Communication Protocols

Remotely manageable malware and more specifically botnets rely on various communication protocols to reach their goals. A botnet is a network of interconnected computers compromised with a malicious software that is controlled by the owner of the network, the bot master. Once infected, the computer joins the botnet and is forced to execute orders received from the bot master such as attacking other computers or sending its personal data. Botnets are known to be responsible for a

1. Skype: `http://www.skype.com`
2. Profibus; `http://www.profibus.com/`
3. BitBus: `http://www.bitbus.org/`
4. Article “List of automation protocols” from Wikipedia: `http://en.wikipedia.org/wiki/List_of_automation_protocols`

large amount of attacks over the Internet. These attacks can take multiple forms, Distributed Denial Of Service (DDOS), large spamming campaigns, financial frauds, Search Engine Optimization (SEO) poisoning, Pay-Per-Click (PPC) frauds, espionage and Bitcoin Mining. Through the years, multiple botnets were revealed by security researchers. A famous example is the Mariposa botnet, one of the largest ones which was responsible for an estimated 13 million infections that were capable of generating at least 250,000€ a month in revenue for the owners [50].

An infected computer or bot, receives his orders from the bot master by means of a specific communication channel called the Command & Control (C&C) channel. The bot master connects to this channel, sends his orders and waits for answers from its bots to gain details over the results of their execution. To evade firewalls and IDSes, C&C protocols are often hidden on top of common network protocols such as IRC, HTTP or P2P protocols. Besides, most botnet protocols are created for a single version of a unique botnet. Obviously, these protocols are kept secrets as they publication would reveal the malware existence along with potential flaws of hidden features in it. This diversity of protocols forces AV and IDS vendors to update their detection signatures every time a new version of a botnet is discovered.

1.2 Motivations

The Reverse Engineering (RE) of communication protocols denotes the processes used to retrieve the specification out of an undocumented protocol. This knowledge of protocol specifications is highly valuable and becomes increasingly important in a number of security-related contexts. For example, security audits of industrial control systems often imply the analysis of proprietary equipment and software. Indeed, an effective security audit of such proprietary system requires to gain sufficient knowledge over the features offered by its protocol. If no documentation is available such as in a black-box audit, the expert is forced to reverse the protocol specification. The importance of automating this process is well established considering the difficulty and time it requires when executed manually.

In addition to security audits and vulnerability assessments, the RE of protocols is also useful for the developers and the evaluators of security products such as Intrusion Detection System (IDS), application firewall, network supervision tool and honeypot. In the following, we outline the three main motivations that led us to focus on the specific field of automated protocol reverse engineering. The importance of protocol RE for security evaluations is detailed in Section 1.2.1. Section 1.2.2 illustrates how protocol RE can be use to create precise NIDS rules. Finally, we detail in Section 1.2.3 why protocol RE is sometimes mandatory to obtain realistic and yet controllable botnet traffic.

1.2.1 Protocol RE for Security Evaluations

Security of Information Systems (SIS) denotes the set of organizational, technical and legal means required to preserve and guaranty the security of Information systems. Among them, the security evaluation of products and systems plays a major role. This process, usually conducted by an independent third party, is designed to check if a product or a system complies against various functional and insurance requirements. In this field, the evaluated product or system is called the Target of Evaluation (ToE) which can latter be certified if the evaluation succeeds. Multiple norms exist to specify the expected security requirements and to guide the evaluation process. The Common Criterias for Information Security Evaluation also known as Common Criteria (CC) [5]) may be one of the best known international framework for the impartial security evaluation of SIS. Other security evaluation frameworks exist such as the French first level security certification (CSPN) [6] and the CESG Assisted Products Scheme (CAPS) [7] in UK.

In recent years, the field of security evaluation of systems or software was extended with new approaches and new tools based on fuzzing techniques. Compared to more traditional techniques (static and dynamic analyzes of the binary, potentially combined with the analysis of the source code) that require specialized skills, resources and time, fuzzing offer many advantages: relative simplicity of implementation, semi-automated approach, rapid acquisition of results, *etc*. However, experience shows that to be truly effective, security analysis by fuzzing requires a good knowledge of the ToE and in particular of the protocol communication that interacts with thereof. This fact limits the effectiveness and completeness of results obtained in the analysis of products implementing proprietary or undocumented protocols.

To evaluate the security of a communication protocol implemented in a ToE, we believe that RE the ToE protocol can be an effective solution. Indeed, protocol RE can be used by security evaluators to gain sufficient knowledge over the product to produce an adapted fuzzer. This fuzzer can latter be used to evaluate the robustness of the protocol implementation.

Another important aspect of an evaluation resides in the assessment of the protocol compliance proposed by the ToE. For example, CC evaluations cover this aspect under the set of tests participating in the Assurance TEsting (ATE) class [23] of its methodology. The objective of this testing class is to provide assurances that the ToE behaves as documented in its functional specification. The RE of the implementation protocol becomes almost mandatory when these tests must be conducted on products of the "secure protocol implementations" category (*e.g.* IPsec, TLS/SSL, EAP, *etc.*).

Indeed, the RE of the ToE protocol makes easier the task of validating protocol compliance. It can be used to produce a protocol model that corresponds exactly to the implementation under evaluation. This model can latter be compared against the theoretical model of the protocol to attest the ToE compliance or to identify deviations [23].

1.2.2 Protocol RE to Build Precise NIDS Rules

A Network Intrusion Detection System (NIDS) is a security application or device that analyzes network communications to detect unauthorized access to a system. IDSes in general, are divided into two families similarly to the classification proposed for IDSes: signature-based and anomaly-based families. A signature-based NIDS searches for specific traces revealing specific threats in the

5. Common Criteria: `http://www.bitbus.org/`
6. CSPN are described at `http://www.ssi.gouv.fr/fr/certification-qualification/cspn/`
7. CAPS are described at `http://www.cesg.gov.uk/servicecatalogue/Product-Assurance/CAPS/`

traffic it analyzes. Its rules or signatures form a collection of all the previously known intrusions. If one rule matches, the alert is investigated to confirm or not the intrusion. On the contrary, an anomaly-based NIDS triggers an alert for every invalid observed protocol usage it observes. Thus, it follows an opposite approach since its rules form a collection of all the accepted patterns of protocol usages.

In both cases, the development costs engendered by these intrusion detection rules are high. The creation of a rule implies to master the threat (or the valid protocol usage) and to transcript it in a language accepted by the product. This mostly manual process is even more complex when it implies unknown or undocumented network protocols. Facing such protocols, most NIDSes rely on a specific bunch of keywords that (may) indicate an intrusion by means of this unknown protocol. Listing 1.1 is an example of a detection rule for Snort IDS [117] that relies on simple keywords, *i.e.* “Wonk-” and “0x00#wate0x00” to detect the P2P Phatbot botnet [32]. This solution is often preferred instead of RE the entire protocol which requires far more work. However, such rule is often responsible of a high number of false-positive as these keywords may also appear in legitimate traffic. Besides, attacks and intrusions tend to constantly evolve to avoid detection. Thus, an attacker may try to evade the NIDS by encoding or modifying its attack thereby effectively changing its signature.

```
alert tcp any any -> any any (msg:"ET P2P Phatbot Control Connection";
   flow: established; content:"Wonk-"; content:"|00|#waste|00|";
   within: 15; [...] classtype:trojan-activity; sid:2000015; rev:6;)
```

Listing 1.1– Snort rule distributed by EmergingThreats company to detect Phatbot botnet C&C

We believe that by reducing the time and costs required to obtain the specification of an unknown protocol, IDS rules developer can develop specific protocol decoders that could be used to develop more robust rules. Thus, RE a botnet protocol could lead to the creation of specific rules that could detect the botnet infection but also the sequence of orders the infected host received from the bot master.

1.2.3 Protocol RE is Mandatory for Effective Botnet Simulation

Computer security history has highlighted the difficulty to have necessary secure systems that would prevent botnets from spreading. Therefore, besides preventing new infections, industry and researchers are also working on the detection of botnets using three families of solutions. The first one composed of Antivirus and Host-based Intrusion Detection System (HIDS) focuses on the malware impact on an infected host [13, 99]. A second family gathers solutions detecting botnets based on a characterization of their proliferation and network topology [36]. The last family considers the presence of a C&C as the main symptom of an infection [57, 27, 143]. This last approach which mostly relies on NIDS rules we previously described seems promising since it targets the major weakness of a botnet: its communications channel. Indeed, once the Command & Control channel is revealed, it becomes possible to prevent the botnet master from controlling its infected hosts.

Even if this approach appears promising, its efficiency has to be validated through the evaluation of its implementation in security tools. To do so, an evaluation methodology has to be expressed, which requires, among other things, the complete qualification of the environment and of the representative dataset used. However, controlling an infected host to obtain a realistic network dataset is not trivial. The difficulty mainly comes from the need to have control over a realistic C&C to validate its detection.

The first and natural way to obtain such dataset is to capture the botnet malware and to use it into an evaluation environment. In this case, the problem comes from the infected host dependencies to one or multiple unmanageable external botnet masters. Indeed, the reproducibility requirement cannot be satisfied when the evaluation environment contains an infected host controlled by an external agent. Moreover, another difficulty brought by the integration of this host into an evaluation process is its aggressive aspect. Besides the generated threats for the evaluation environment, an infected host is a threat for other hosts through its participation in distributed malicious operations. In addition, malware also often include protections against reverse engineering tools and anti-virtualization procedures. Hence, an evaluation process including connected infected hosts is often very expensive and non-reproducible. These limitations justify the use of network traffic generators instead of real botnets.

Network traffic generators can be divided into two main categories. The first one covers all the **replay solutions** that re-inject a captured traffic (e.g. pcap files) obtained from an existing C&C in the evaluation network. This approach can be easily and rapidly implemented in an evaluation process. However, in addition to the lack of privacy for the actors involved in the dataset, the injected traffic can also introduce out-boundaries behaviors such as uncharacterized attacks, incompatible protocols and outdated values. Therefore an expensive preparation step must be accomplished upstream to analyze the captured traffic [18]. The second category of network generators regroups all the **synthetic solutions** and produces traffic based on heuristics and published statistics [92]. These statistics model the evolution over time of an actor's behavior. However, generated traffic is often too simple and unrealistic. Indeed, synthetic models used to generate network traffic cannot effectively address all the specifics of the environment in which the botnet evolves. To address these issues, we proposed an **hybrid solution** that relies on the RE of a botnet protocol to generate a realistic and controllable dataset [25]. Indeed, the inferred model of a botnet protocol can be shared among security evaluators. Based on this model, an evaluator can thereafter generate a realistic dataset that can be used during evaluations.

In this Section, we exposed three security-related contexts that could be improved by means of an automated reverse engineering technique. We claim that security evaluations can take advantage of a RE technique that produces high quality specifications of the protocol used by an implementation. We claim that a fast RE technique would help developers of NIDS rules to be more responsive in their work against computer threats. Finally, we claim that a stealthy RE solution can be used to infer botnet protocols for the creation of realistic and yet controllable botnet traffic generators. For all these reasons, this thesis addresses the need in an effective, stealthy and yet fast automated reverse engineering approach that applies on communication protocols.

1.3 Thesis Statement and Contributions

In regards to the number of protocols available and, among them, the number of undocumented ones, we decided to investigate the field of protocol Reverse Engineering. Our motivations comes from the need of protocol specification in numerous fields and from our observations of the lack of practical and effective approaches to obtain these specifications by RE.

Problem Statement:

Current work in the field of automated protocol reverse engineering either infer incomplete protocol specifications or require too many stimulation of the targeted implementation with the risk of being defeated by counter-inference techniques. Besides, none of these work infer detailed enough specification that could support the simulation of the reversed protocol. Finally, the computation time required by previous RE work can be enormous which can prevent their uses where high responsiveness is mandatory.

We extract from our problem statement, the three following objectives of our work:

Objective 1 Our protocol RE solution must produces precise, correct and complete protocol specifications that models both the vocabulary and the grammar of an undocumented protocol.

Objective 2 Our protocol RE solution must be faster than existing work.

Objective 3 Our protocol RE solution must increase the stealthiness of the inference process in comparison to previous work.

To attain these objectives, we expose the following thesis statement.

Thesis Statement:

The semantic definition of a protocol can be used to improve the syntactical and grammatical inference of a communication protocol. Leveraging the protocol semantic in an automated reverse engineering approach improves the overall quality of the inferred protocol specifications, reduces the computation time and increases the stealthiness of the inference process.

Thesis Contributions:

This thesis proposes a practical approach for the automatic reverse engineering of undocumented communication protocols. This work leverages the semantic of the protocol to improve the quality, the speed and the stealthiness of the inference process when applied on complex protocols. Our work covers the two main aspects of the protocol RE, the inference of its syntactical definition (the protocol vocabulary) and of its grammatical definition (the protocol grammar). The algorithms we propose uses the semantic definition in both domains. We conducted multiple experiments to validate our approach by comparing previous state-of-the-art work against our algorithms. We also propose an open-source tool, called Netzob [8], that implements our work to help security experts in their protocol reverse engineering tasks. We claim Netzob is the most advanced published tool that tackles issues related to the reverse engineering and the simulation of undocumented protocols.

8. Netzob: `http://www.netzob.org`

The contributions on this dissertation are summarized in the following:

— We introduce a new trace-based approach to infer the vocabulary of a protocol. It leverage various semantic information in different pre-computing steps to enhance protocol vocabulary inference.

— We propose a correlation-based approach to automatically infer relationships between message fields.

— We detail a parallel approach to reverse the grammar of an unknown protocol that drastically reduces the inference time.

— We propose a grammatical inference process that leverages semantic information for a stealthier reverse engineering.

— We expose a solution that infers the reaction time of protocol implementation to increase the realism of the inferred model.

— We present the results of an experimental comparative study that compares our work against other state-of-the-art solutions in the field of protocol vocabulary and grammar reverse engineering.

— We publish an open-source and freely available tool that implements our algorithms to reverse both the vocabulary and the grammar of protocols.

1.4 Dissertation Outlines

This dissertation is organized as follows. First, some insights on common communication protocols are provided in Chapter 2. These examples are followed by a more formal definition of a communication protocol. Chapter 3 details state-of-the-art works in the field of protocol reverse engineering. This study highlights the main issues identified by these pieces of work and the solution proposed relatively to these. We conclude this chapter with a discussion on the recurrent issues in the field of RE and with a summary of the available tools. Our dissertation is then divided into two parts. Part I presents our contributions in the field of vocabulary inference and Part II our contributions in the field of grammatical inference. These parts are organized as follows:

Part I - Automated Inference of the Protocol Vocabulary presents our solution to infer the vocabulary of undocumented protocols by means of an automated approach and novel techniques that leverage protocol semantic. Based on communication traces, we reverse the vocabulary of a protocol by considering embedded contextual information. We also use this information to improve message clustering and to enhance the identification of fields boundaries. We then show the viability of our approach through a comparative study including our re-implementation of three other state-of-the-art approaches. Part I comprises three chapters:

— **Chapter 5 - Our Vocabulary Model** covers our definition of a symbol, of its fields and of their definition domains including optional relationships among them. Then, we describe the abstraction and specialization processes we use to transform symbols into contextualized and syntactically valid messages. Finally, this chapter presents the memory mechanism we use to support relationships between fields such as size fields or sequence numbers.

— **Chapter 6 - Leveraging Semantic Information to Improve the Vocabulary Inference** describes the solution we retained to infer the vocabulary of a protocol out of sample traces. Specifically, after a high-level overview of our methodology, it details the two main parts of our automated reverse engineering process: a semantic-based message clustering and the Field Relationships Identification.

— **Chapter 7 - Comparative Study of Vocabulary Inference Approaches** exposes the evaluation of our approach to infer the vocabulary of various protocols and compares our contributions against state-of-the-art approaches. Two different kinds of experiments are conducted: 1) on known protocols to compare inferred message formats with their published specifications and 2) on unknown protocols to evaluate the effectiveness of the different approaches on more operational use cases. This chapter gives some key insights over the compared tools and then present the datasets, the metrics and the implementations we used in this study. It concludes with a discussion on obtained results.

Part II - Automated Inference of the Protocol Grammar details our work in the field of grammatical inference. It describes our solution that leverage contextual information and semantic definition associated with protocol usages as key parameters in the grammatical inference of a protocol. It shows how we rely on this semantic information to split the large inference task into separate parallel sub-tasks which drastically reduces the computation time of the whole inference. It also explains that our solution reduces the stimulation of the inferred implementation thus being stealthier. Part II comprises four chapters:

— **Chapter 10 - Our model of a Protocol Grammar** describes the Symbolic Mealy Machine (SMM) we use to model the grammar of a protocol. It consists in an extension of a Mealy Machine that supports a symbolic vocabulary along with the definition of a reaction time for each transition it denotes. This timing data models the average elapsed time between the emission of a message and its associated response. We also describe in this Chapter the advantages that arise with our idea of decomposing a protocol grammar to improve its inference.

— **Chapter 11 - Learning the Grammar Using an FSM Decomposition** details our grammatical inference process. Specifically, after a high-level overview of our methodology, this chapter describes how we identify and leverage protocol features to parallelize the inference process. It also explains the algorithm we use to retrieve the protocol grammar out of partial sub grammars inferred in parallel.

— **Chapter 12 - Evaluation** is a comparison of our results against those computed by the classical version of the state-of-the-art L^* algorithm exposed by the LearnLib [111] framework. This evaluation consists in three different experiments, each applied on a different protocols. Among the retained protocols, two are famous known protocols while a last one is an undocumented protocol used by a botnet. This comparison shows that our approach is effective to compute a good approximation of the targeted protocol grammar while being faster and stealthier than previous work. We conclude this chapter with a discussion on obtained results.

Chapter 2

Communication Protocols

In this chapter, we describe the foundations of the communication protocol research fields. In section 2.1 we overview recurrent forms and usages of communication protocols through two protocols: 1) a common text protocol and 2) a recent P2P botnet protocol. In section 2.2, we rely on Gerard Holzmann's work [65] to give a more formal definition of a communication protocol and of its two main components: the vocabulary and the grammar. We then present existing formal languages and techniques used for protocol specifications in section 2.3.

2.1 Recurrent Forms of Communication Protocols

A communication protocol is a standard set of digital rules governing information exchanges between actors. These rules can be as simple as the introduction of keywords to support conversation in morse-based languages [5], to highly complex as in TCP/IP protocols over which the Internet works. Besides the large variety of protocol usages, rules that govern protocols express three recurrent key features: 1) a communication establishment scenario, 2) the information exchanges and 3) how to deal with errors. In addition to these key features, protocols often include additional rules to support properties such as confidentiality as provided by the IPsec suite of protocols [77] and extended fault-tolerances processes as in Train Communication Networks [81].

Obviously, to ensure that both the sender and the receiver of a communication follow the same rules, protocols specifications must be shared. As noted by G. Holzmann [65], the IBM Bi-SynC protocol (BSC) [68] and its chaotic "enhancements", *i.e.* more than fifty incompatible variants, has revealed the necessity for international standards in the field of protocol specification. Such standard aims at ensuring a uniform adoption of protocols among all the constructors and software developers. To achieve this, many international standardization bodies exist to harmonize the technical specifications of protocols. Among them, the Institute of Electrical and Electronics Engineers (IEEE) and the International Standards Organization (ISO) [1] are the most important. In the field of protocol standards, the Internet Engineering Task Force (IETF) [2] develops and promotes Internet standards through the publications of memorandum called Request for Comments (RFC).

1. ISO's website: `http://www.iso.org`
2. IETF's website: `http://www.ietf.org`

Another important actor in this field is the International Telecommunication Union (ITU)[3], a specialized agency of the United Nations that is responsible for issues that concern information and communication technologies including the coordination of worlwide technical standards.

In the remainder, we give some insights over the definition of protocols used in two different kind of communications: Internet related protocols and botnet protocols. The first case study covers the Hypertext Transfer Protocol 1.1 [49] as the most used protocol over Internet [21, 84, 141]. The second case study analyzes the peer-to-peer communication protocol used by a recent malware: the ZeroAccess botnet [22]. With these case studies, we explore common protocol features regarding their respective environment and usages to search for common underlying principles.

2.1.1 HTTP Case Study: A Common Text Application Protocol

The Hypertext Transfer Protocol (HTTP) and its most notable version HTTP/1.1 published in 1999 [49], is the result of a coordinated work between the IETF and the World Wide Web Consortium (W3C). Its standard describes a text based request-response protocol, in use between a client and a server to exchange application data on top of TCP [109]. Internet navigators, such as Firefox or Internet Explorer, use this protocol to download website content during an Internet navigation. The navigator plays the role of an HTTP client and sends HTTP requests to a website hosted by an HTTP server that answers with HTTP responses. HTTP is a stateless protocol that only accepts sequences of messages that follow a request/response pattern initiated by the client.

An HTTP request denotes a specific method (*e.g.* `GET`, `HEAD`, `POST`, `PUT`) indicating the desired action to be performed on a given resource. Some of these methods are only intended for information retrieval, such as the `HEAD`, `GET`, `OPTIONS` and `TRACE` methods while others may change the server internal state such as `POST`, `PUT` or `DELETE` methods. A request message has a specific format that consists of a “Request-Line” followed by a “Request Header” and a message body.

The Request-Line is made of three successive fields separated by a space character. The first field contains the method name also called the request command. Its value must be one of the following: `GET`, `HEAD`, `POST`, `OPTIONS`, `CONNECT`, `TRACE`, `PUT`, `PATCH` or `DELETE`. The next field denotes the resource on which the command applies and must be an URL. Finally, the third field of the Request-Line contains the protocol version number.

The Request Header consists in a set of name-value pairs, each pair denoting a property. Properties are separated with a CRLF sequence of characters. HTTP specifications describe a property under an ABNF language as illustrated on Listing 2.1. It specifies it as the concatenation of a token, the “:” character and a value. Previously in specification, a token is defined as a string that contains neither American ASCII control characters or delimiters. The value property is made of any sequence of printable ASCII characters, token, separators, space character or quoted string. The only mandatory property that must be present in a Request Header is the Host property that denotes an hostname optionally followed by a port number (*e.g.* “`www.w3c.org:80`”).

The message body is separated from the previous field with a blank line. It is used to carry the

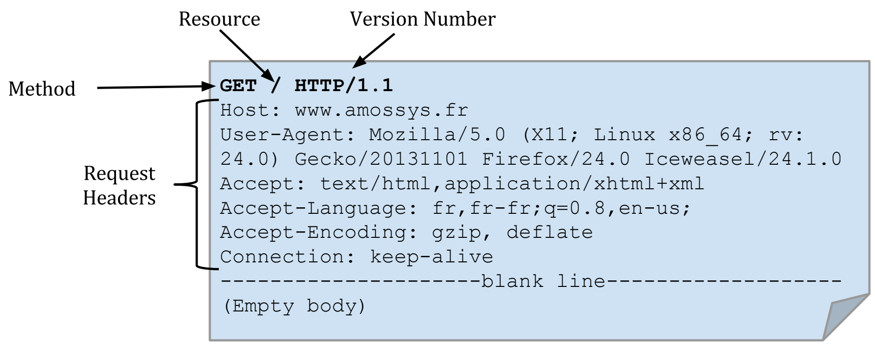
3. ITU’s website `http://www.itu.int`

entity-body associated with the request or the response message. If a message-body is specified, the Request Header must include a Content-Length or a Transfer-Encoding property. Figure 2.1 illustrates an example of an HTTP request.

```
message-header = field-name ":" [ field-value ]
       field-name     = token
       field-value    = *( field-content | LWS )
       field-content  = <the OCTETs making up the field-value
                        and consisting of either *TEXT or combinations
                        of token, separators, and quoted-string>
```

Listing 2.1– ABNF definition of the HTTP header properties as described in RFC 2616 [49])

HTTP response messages accept a very similar message format. It denotes a "Status-Line" field made of the server version number and a status identifier that contains a numeric status code, such as "200" and a textual reason phrase, such as "OK". This line is then followed with properties stored in a "Response Header". An optional message body can also follow.



```
GET / HTTP/1.1
Host: www.amossys.fr
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:
24.0) Gecko/20131101 Firefox/24.0 Iceweasel/24.1.0
Accept: text/html,application/xhtml+xml
Accept-Language: fr,fr-fr;q=0.8,en-us;
Accept-Encoding: gzip, deflate
Connection: keep-alive
---------------------blank line-------------------
(Empty body)
```

Figure 2.1 – Sample HTTP `GET` request with highlighted fields.

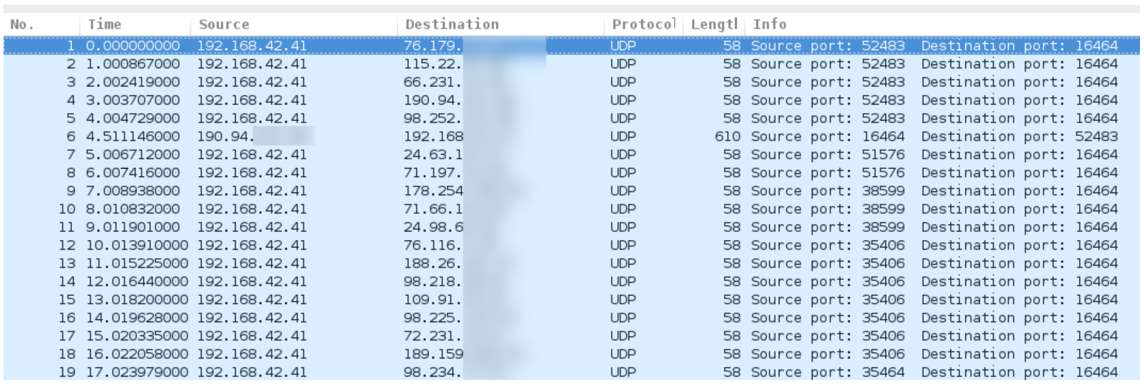
To summarize, HTTP is a stateless text-based protocol that enables data exchanges on top of TCP. It follows a request-response communication pattern always triggered by the client. An HTTP message consist in a header made of multiple fields followed by an optional payload that can host data brought by protocols on top of HTTP. Messages can be classified in different types following the semantic they denote. However, all the request messages share the same protocol format and the type information is represented by one of its field value. Similarly, all HTTP response messages follow the same field definition. Regarding its format, HTTP makes an heavy use of ASCII delimiters (*e.g.* "␣", ":", "CRLF") and very few size fields such as the "Content-Length" field. Besides, some of its fields are optional and no specific rule establishes their declaration order. Finally, it exists very few relationships between its fields value. Many communication protocols that belongs to the highest layers of the ISO model share similarities. Indeed, traditional protocols belonging to these layers such as the application layer were often created with an objective of being usable and readable by humans. It explains the use of ASCII to encode exchanged data. For example, the SMTP [110, 79], FTP [62] and IRC [102, 74] protocols share similar message formats with their ASCII fields delimited with specific ASCII characters.

2.1.2 ZeroAccess Case study: A P2P Botnet Protocol

ZeroAccess is a recent botnet discovered around July 2011 by Symantec that infects Windows operating systems [123]. Its primary motivations is to make money through Bitcoin mining and pay-per-click advertising. Its size has been estimated at around one million active on at least nine million systems in the third quarter of 2012.

The malware spreads itself through various attack vectors. Among them, ZeroAccess was found in apparently legitimated files that users download from infected websites. It also relies on classical sets of "drive-by-download" attacks distributed by the Blackhole Exploit Toolkit and the Bleeding Like Toolkit [66]. Once executed on a computer, this malware behaves as a typical rootkit to hide and persist on the compromised system. Typically, it infects the Master Boot Record (MBR) of its host and disables the Windows Security Center service and with it, the user firewall and anti-virus provided by Windows 7. It also downloads other malware and lure the user to download fake anti-viruses applications.

Moreover, it opens a backdoor to connect to its network. Its command and control channel is used to distribute updates and malicious files among all the botnet members. ZeroAccess has seen multiples updates. In the following, we focus on the C&C protocol it operates after its update on the second quarter of 2012. From our knowledge, latest observed protocol updates occurred the 29 of June 2013 which included small improvements.



```
No.  Time         Source          Destination  Protocol Lengtl Info
   1 0.000000000  192.168.42.41   76.179.      UDP       58 Source port: 52483  Destination port: 16464
   2 1.000867000  192.168.42.41   115.22.      UDP       58 Source port: 52483  Destination port: 16464
   3 2.002419000  192.168.42.41   66.231.      UDP       58 Source port: 52483  Destination port: 16464
   4 3.003707000  192.168.42.41   190.94.      UDP       58 Source port: 52483  Destination port: 16464
   5 4.004729000  192.168.42.41   98.252.      UDP       58 Source port: 52483  Destination port: 16464
   6 4.511146000  190.94.         192.168      UDP      610 Source port: 16464  Destination port: 52483
   7 5.006712000  192.168.42.41   24.63.1      UDP       58 Source port: 51576  Destination port: 16464
   8 6.007416000  192.168.42.41   71.197.      UDP       58 Source port: 51576  Destination port: 16464
   9 7.008938000  192.168.42.41   178.254      UDP       58 Source port: 38599  Destination port: 16464
  10 8.010832000  192.168.42.41   71.66.1      UDP       58 Source port: 38599  Destination port: 16464
  11 9.011901000  192.168.42.41   24.98.6      UDP       58 Source port: 38599  Destination port: 16464
  12 10.013910000 192.168.42.41   76.116.      UDP       58 Source port: 35406  Destination port: 16464
  13 11.015225000 192.168.42.41   188.26.      UDP       58 Source port: 35406  Destination port: 16464
  14 12.016440000 192.168.42.41   98.218.      UDP       58 Source port: 35406  Destination port: 16464
  15 13.018200000 192.168.42.41   109.91.      UDP       58 Source port: 35406  Destination port: 16464
  16 14.019628000 192.168.42.41   98.225.      UDP       58 Source port: 35406  Destination port: 16464
  17 15.020335000 192.168.42.41   72.231.      UDP       58 Source port: 35406  Destination port: 16464
  18 16.022058000 192.168.42.41   189.159      UDP       58 Source port: 35406  Destination port: 16464
  19 17.023979000 192.168.42.41   98.234.      UDP       58 Source port: 35464  Destination port: 16464
```

Figure 2.2 – UDP traffic generated by a host infected by ZeroAccess

The C&C protocol of the ZeroAccess botnet is a P2P protocol. It enables the creation of a distributed directory of all the infected hosts by means of UDP connections. This directory is used by each bot to identify from which other peers it can download malicious files or updates. This protocol does not cover files transfer. As illustrated on Figure 2.2, an infected host constantly contacts other peers to update its peer list and to discover new files to download. As a matter of facts, each bot is also constantly contacted by other peers. Thus, a ZeroAccess bot plays both the role of a server and a client.

To avoid easy detection, each message is encrypted by means of a rotated XOR. It encrypts (or decrypts) four-byte at a time the message using a four-bytes key. The initial key value is "ftp2". The routine given in Listing 2.2 can be use to decrypt ZeroAccess communications. Its protocol vocabulary is made of three different types of binary message (*getL*, *retL* and *newL*). In the following, we detail their formats.

```
import struct

def decryptZeroAccessMessage(encryptedMessage):
    key=0x66747032
    result = []
    for i in range(0,len(encryptedMessage), 4):
        subData = struct.unpack("<I", encryptedMessage[i:i+4])[0]
        xoredSubData = subData ^ key
        result.append(struct.pack("<I", xoredSubData))
        key = ((key << 1) & 0xffffffffL | key >> 31)
    decryptedMessage = ''.join(result)
    return decryptedMessage
```

Listing 2.2– Python decryption routine of ZeroAccess messages

The *getL* message is the first message an infected host emits to a predefined list of peers. With this message, the infected host requests a new list of peer IP addresses. As illustrated on Figure 2.3, a *getL* message is made of four fields of four bytes long. The first field contains the message CRC32 value and the second field, the message command name (*i.e.* “getL”). The third field is filled with zeros while the last field contains a randomly generated number, the bot unique identifier.

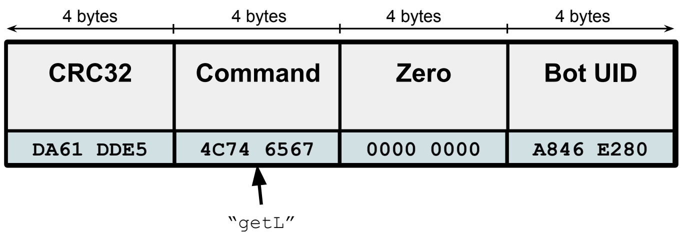


Figure 2.3 – ZeroAccess *getL* message format

The *retL* message is another type of message that is sent in response to a *getL* message. Figure 2.4 illustrates its format. It contains a list of IP addresses of other botnet members and a list of files that can be downloaded. Similarly to the *getL* format, the first and the second field host the CRC32 value and the command name of the message. Obviously, in this case the second field is always filled with the “retL” value. The four-byte value stored in the third field is often referred to as the “broadcast flag” that might indicate if the receiver must propagate the list of IP addresses contained in this message to its own peers. The fourth field contains the number of IP and timestamp pairs that are stored in the fifth field (denoted “IP Entries” on Figure 2.4). Each pair consists in two values of four-bytes: the IP address of a peer and its timestamp. Right after this sequence of IP/timestamp pairs, the sixth field denotes the number of file entries contained in the last field (denoted “File Entries” on Figure 2.4). A file entry is made of four values. The first value denotes a file name of four bytes long followed with the file creation date also a four-byte long value. The third value in a file entry denotes the file size while last value is a 32 bytes long that might represent the file signature.

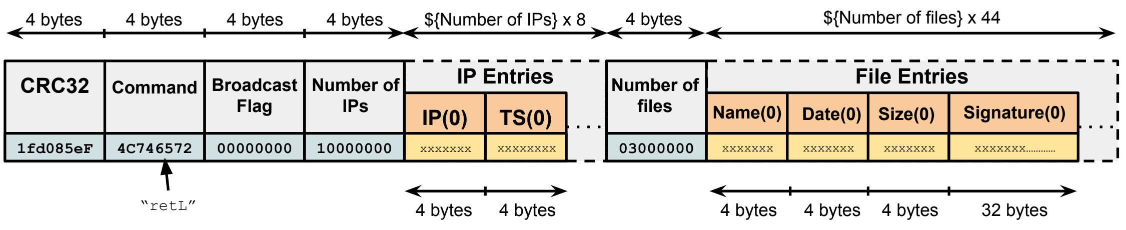


Figure 2.4 – ZeroAccess *retL* message format

When a bot receives a *retL* message it checks the file names and creation dates declared in it against the files it has. If it discovers that the remote peer possesses a file it does not have, it tries to obtain a copy of it. To achieve this, it initiates a TCP session to the peer on the same port number as the UDP exchange and downloads the file by means of another protocol.

The *newL* message propagates a new peer address across the botnet. When an infected host receives a *retL* message with the broadcast flag set, it broadcasts the received peer list to its own peer list through a set of *newL* message. This message follows a similar format than the *getL* message. As illustrated on Figure 2.5, a *newL* message is made of four fields. The first and second field respectively contains the CRC32 value and the command name ("newL") of the message. The meaning of the third field is obscure and usually contains "8". The fourth field contains the peer IP address the sender wants to propagate.

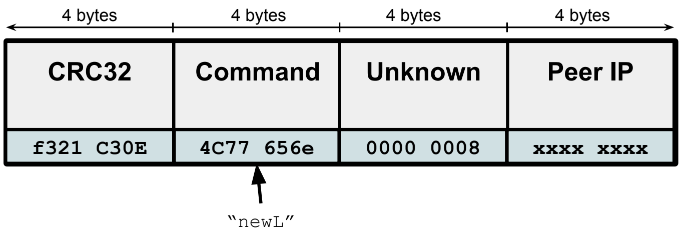


Figure 2.5 – ZeroAccess *newL* message format

Zero Access P2P protocol relies on binary messages. These messages can be regrouped in three types following the value of their second field. Based on this value, the parser expects a specific format to parse the remaining data. This format is made of a mix of static sized fields and fields with a size computed following previously parsed field values. Another interesting thing is its encryption. Its objective is not to ensure the confidentiality of its exchanges but rather to prevent its detection through signature based mechanisms. As detailed in [22], reverse engineer this protocol can be easily achieved if a preliminary crypto analysis is performed to break the encryption mechanism. It requires to identify field boundaries and to cluster messages having a similar format.

2.2 Formal Definition of a Communication Protocol

A communication protocol can be defined as the set of rules allowing one or more entities (or actors) to communicate. Applied to the field of computer networking, protocols have been the subject of many standardization activities, particularly from the OSI model, which establishes, among other things, the principle of protocol layers. However, few studies have attempted to give a formal and generic definition of a communication protocols. We refer here to the definition provided by G. Holzmann in his reference book *Design and Validation of Computer Protocols* [65]. According to the author, a protocol specification consists of five distinct parts:

1. The *service* provided by the protocol.
2. The *assumptions* about the environment in which the protocol is executed.
3. The *vocabulary* of messages used to implement the protocol.
4. The *encoding (format)* of each message in the vocabulary.
5. The *procedure rules* guarding the consistency of messages exchanges.

In our work, this specification is unknown and we try to infer it from observed messages using an implementation of the protocol. As pointed by G. Holzmann, the fifth part is the hardest to develop and to verify. In our case, it is also the most difficult to infer. Furthermore, this definition is generic, somehow “fractal”, since each part can define its own hierarchy of elements. For example, message format can define additional embedded messages. This corresponds to the notion of protocol layers defined in the standards listed above.

In this thesis, we seek to infer the three last elements of the specification. Subsequently, we consider protocol inference requires to learn both, 1) the set of messages and their format, also called the **vocabulary** and the **syntax** of the protocol and 2) all the rules of procedure that we name **grammar** of a protocol. We give a more formal definition of these notions in the rest of this section.

2.2.1 Definition of the Protocol Vocabulary

As presented below, the definition of a protocol is similar to the one of a language, and as so, includes a vocabulary and a syntax [65]. The vocabulary lists all the valid messages of the protocol while the syntax, also called the message format, denotes the rules and principles by which messages are constructed [37]. For example, the ICMP protocol denotes a vocabulary composed of messages such as echo requests or echo responses and a protocol syntax that defines the fields structure of these messages.

Protocol Vocabulary

The message vocabulary of a protocol, also called *protocol vocabulary*, lists the messages that can be exchanged by the actors of a communication. Besides its format covered by the protocol syntax, a message denotes one or more meanings, *i.e.* its semantic, and therefore implies a specific impact in a sequence of exchanged messages, *i.e.* one message cannot be replaced by the other without changing the objectives of the exchange. For example in the TCP protocol, `SYN` messages cannot be replaced by `ACK` messages without breaking the three-way handshake. That is because both messages have a different meaning in the TCP protocol, *i.e.* `SYN` messages indicate a connection establishment request while `ACK` messages indicate an acknowledgment. In the remainder, we refer to the type of a message to denote both the semantic and the syntax of a message. For example, the XMPP instant-messaging protocol, as described in RFC 6121 [119], also exposes various message types such as presence messages, chat messages or roster request messages.

Nevertheless, two messages of the same type can be different. In effect, messages often include parameters such as IP addresses, nicknames, serial numbers or message identifiers. Certain parameters can take their values in a theoretically infinite definition domain or can depend on the value of others. Consequently, there can be an infinite number of messages of a single type. For example, the `RETR` message sent by an FTP client to retrieve a remote file includes as parameter, a filename. Thus, an FTP capture can contain a large amount of different `RETR` messages denoting variations introduced by the filename parameter. However, despite their orthographic differences all the `RETR` messages share the same meaning and message format.

Figure 2.6 illustrates another example of two messages of the same type that are different. In this example, the two messages were collected in the C&C of the TDL botnet [56]. Even if they differ, both can be associated to the same symbol as they are periodically sent to the botnet master to retrieve the available commands [116]. Their differences come from the value of their nested parameters such as a random number and a bot identifier number that are specific to the context.

Message 1
```
command|6c23-1261-A2987381|40379|0|3.23|0.15|5.1 2600 SP3.0|en-us|iexplore ...
```

Message 2
```
command|1b4304f0-66a4-153d|10616|0|3.23|0.15|5.1 2600 SP3.0|en-us|iexplore ...
```

Figure 2.6 – Anonymized example of two TDL bot requests.

To represent the vocabulary of a protocol in a more compact and organized model, most works [20, 17, 2] in the field of protocol reverse engineering, use a **symbolic vocabulary**. In such approach, same-type messages are replaced by a single abstraction called a **symbol** in which parameters are identified and replaced by their definition domain. This definition domain defines all the valid values that could be taken by the parameters. Indeed, a symbol can be defined as the common abstraction of multiple messages, sharing a common syntax and having the same role from a protocol perspective. For example, the SMBv2 official specification[4] defines symbols such as `SESSION_SETUP` or `LOGOFF_REQUEST` that drive user authentication exchanges.

Protocol Syntax

As presented below, a symbol represents a set of messages that share the same syntax and the same semantic. The syntax of a message, also called the message format, defines the rules under which messages are built [37]. These rules establish the valid sequences of words that compose each message. Its definition is somehow "fractal" because its basic unit, the word, also refers to a sequence of letters (or bits). For sake of clarity, we focus on syntactic rules that govern a sequence of words and let the reader apply this definition on the inner-composition of these words. In the following, we define the notion of word when applied on computer-related communication protocols.

4. SMBv2 specifications: `http://msdn.microsoft.com/en-us/library/cc246497.aspx`.

Finally, we describe recurrent syntactic rules used by protocols to describe words composition in a message.

A word consists in a sequence of bits that are significant as a group. For example, the sequence of bits under the ASCII "127.0.0.1" denotes an IP address. "192.168.0.1" is another word that also represents an IP address. As illustrated on Figure 2.7, we can split a message such as "GET index.html" into four words: "GET", "␣", "index" and ".html" where first word denotes an action, the second a delimiter, the third a file name and the fourth a file extension.

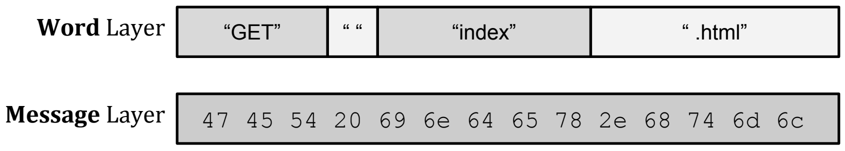


Figure 2.7 – A message can be split into words.

A word is related to a lexical item called a token, *i.e.* an abstract unit, that denotes a basic unit meaning. A token abstracts all the possible words that share its meaning. That is to say that a word reflects a specific orthographic definition while its associated token denotes its meaning. For instance, "10.20.30.40" and "10.11.12.13" are two different words as their sequence of bits differ. However, they refer to the same semantic definition, *i.e.* an IP address. They can therefore be abstracted by the same token. Thus, a token is what a symbol is for a message but applied to words: it denotes the meaning and the format shared by various orthographic variation of the same type of word. Figure 2.8 illustrates the token definitions related to each word participating in our previous example.

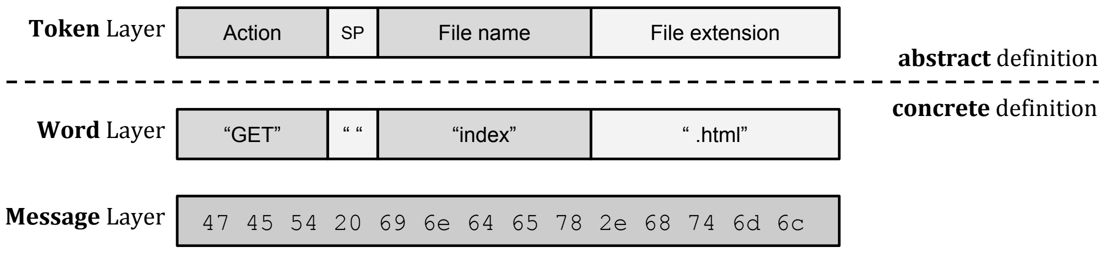


Figure 2.8 – A message can be split into words that are related to tokens.

Tokens can take various forms following their content but two main categories of token arise: **text tokens** and **binary tokens**. Text tokens are made of ASCII or any Unicode related characters as in HTTP messages while binary tokens denotes a sequence of bits as in DHCP messages. In presence of text tokens, specific characters (*e.g.* ":", "␣", ";") or sequence of characters (*e.g.* "CRLF") can be used as delimiters. On the other hand, delimiters are rarely used to separate binary tokens. Instead tokens have either a static known size (*e.g.* 2 bytes, 4 bytes) or a dynamic size computed following other token values. In some protocols, messages use a unique type of token, either text or binary ones. This consistency mostly comes from the complexity of having either a complex generic parser that supports both text and binary tokens or to change the parser at runtime.

A protocol that only denotes text tokens is called a text protocol while only binary tokens produce binary protocols. It also exists various protocols such as the DNS protocol that embeds both text and binary tokens.

In addition, a token can either denotes a static word, *i.e.* its value never change across all the possible messages, or dynamic, *i.e.* the same token in two messages of the same symbol generates different words. For example, the HTTP specifications defines the `HTTP REQUEST` symbol with both static and dynamic tokens (listed in table 2.1) This definition accepts various messages such as "GET /etude.php HTTP/1.1" or "PUT /form.php HTTP/1.1".

Token Name	**Token Variation**	**Token Values**
Method	Dynamic	"OPTIONS" or "GET" or "POST" or ... or "CONNECT"
SP	Static	" "
Request-URI	Dynamic	"*" or an absolute URI
Version-Header	Static	"HTTP/"
Version-Major	Dynamic	Positive Integer
Dot	Static	"."
Version-Minor	Dynamic	Positive Integer

Table 2.1 – Tokens participating in the specification of the "Request-line" in `HTTP REQUEST` symbol.

As in our previous HTTP example, multiple tokens can participate in the definition of a symbol. To define the valid sequence of tokens, a symbol uses syntactic rules that establish the underlying symbol format. These rules are very similar to the one that produce valid sequences of symbols and as so can be regrouped under a grammatical form. However, in very most protocols the definition of valid token sequences, that represent a symbol, relies on a normal disjunctive form[5] defined with two basic sequence operations: aggregate (*i.e.* a concatenation of zero of more tokens) and alternate (*i.e.* a possible set of expected tokens). For example, the request-line of the `HTTP REQUEST` symbol is defined as an aggregation of the following tokens: Method, SP, Request-URI, SP, Version-Header, Version-Major, DOT, Version-Minor and CRLF. Another common operation is the optional repetition of these operations. Each protocol specification language proposes a specific set of basic sequence operations which can be used to express more complex symbol format. We detail existing specification languages, that can be use to define token sequences for a protocol, in section 2.3.1.

As explained below, the format of a symbol is defined in terms of tokens. However, most protocol specifications relies on an intermediate abstraction unit, called **field**, to represent one or more consecutive tokens participating in the same semantic. For instance, among all the fields that participate in the specification of a standard DNS query illustrated on figure 2.9, the "Name" field is made of three tokens: a domain name, a delimiter and a domain suffix. Figure 2.9 summarizes

5. Disjunctive normal form. Encyclopedia of Mathematics: `http://www.encyclopediaofmath.org/index.php?title=Disjunctive_normal_form&oldid=14566`

the definition of a symbol. It illustrates that a symbol is made of a sequence of **fields** each being composed of one or more lexical **tokens**. Finally, these orthographic words compose a final message.

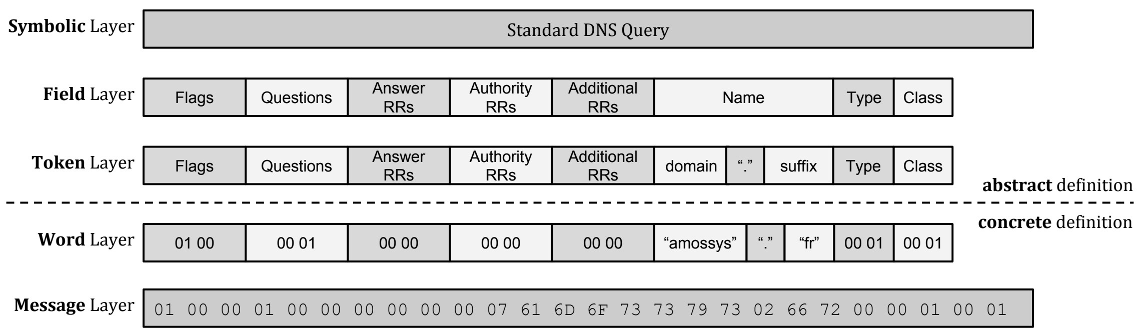


Figure 2.9 – Illustration of the abstraction layers participating in the specification of a standard DNS query.

2.2.2 Definition of the Protocol Grammar

We detailed the notion of vocabulary and of its basic units (words, tokens, fields, symbols) in section 2.2.1, we now focus on the definition of valid protocol exchanges. By exchange, we refer to the ordered sequence of sent and received symbols between actors of a communication. For example, the following sequence of symbols `[SYN, SYN/ACK, PUSH, ACK, FIN/ACK, ACK]` is a valid TCP exchange whereas `[ACK, SYN, FIN/ACK, PUSH]` is not.

To define these exchanges, a first naive approach consist in the use of a list of all the valid sequences of symbols. However this solution does not apply with infinite languages which obviously would require an infinite memory. Typically, instant-messaging protocols rarely limit the number of possible exchanges between users. Therefore, protocols express this list in a more compact way using dedicated rules such as "symbol 3 always follows symbol 2" or "symbol 1 can not be consecutively repeated". The advantage raises mainly from the usage of a small number of rules to represent a large number of valid exchanges. This set of rules denoting all the valid sequences of symbols is called the protocol grammar.

Formerly, a grammar is defined by a 4-tuple $G = \langle V, \Sigma, P, S \rangle$, with V the set of symbols representing a subset of the language also called the **nonterminal symbols**, Σ a finite set of symbols that can occur in the final sequence of symbol also called the **terminal symbols**, P the finite set of Production Rules (PR) that transform nonterminal symbols into terminal symbols and $S \in V$ the start symbol used to represent the whole sentence. These production rules are the key aspect of each grammar definition as they transform nonterminal symbols into a sequence of either terminal, nonterminal or empty symbols. They are of the form $V \to (V \cup \Sigma)^*$.

A typical example of a grammar is $G = \langle V = \{S\}, \Sigma = \{\texttt{LOGIN}, \texttt{LOGOUT}\}, P, S \rangle$ with P defined with PRs listed in 2.3. This grammar generates a language that describes all nonempty exchanges of `LOGIN` and `LOGOUT` symbol that ends with `LOGOUT`. For instance, one of a typical derivation of this grammar produces the sequence [`LOGIN`, `LOGOUT`, `LOGIN`, `LOGOUT`] by successively applying production rules `PR2`, `PR3`, `PR3`, `PR1` on the start symbol S ($S \rightarrow [\texttt{LOGIN}, S] \rightarrow [\texttt{LOGIN}, \texttt{LOGOUT}, S] \rightarrow [\texttt{LOGIN}, \texttt{LOGOUT}, \texttt{LOGIN}, S] \rightarrow [\texttt{LOGIN}, \texttt{LOGOUT}, \texttt{LOGIN}, \texttt{LOGOUT}]$).

```
PR1: S → LOGOUT
PR2: S → LOGIN,S
PR3: S → LOGOUT,S
```

Listing 2.3– PR for nonempty sequences of `LOGIN` and `LOGOUT` symbols ending with `LOGOUT`.

The Chomsky Hierarchy of Grammars

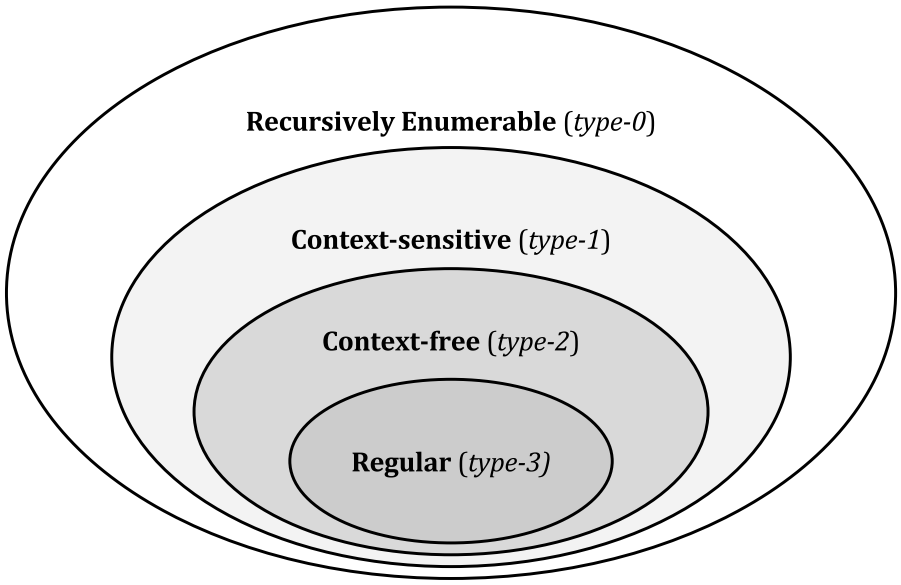

Recursively Enumerable (*type-0*)
Context-sensitive (*type-1*)
Context-free (*type-2*)
Regular (*type-3*)

Figure 2.10 – Chomsky Hierarchy

Numerous types of grammar exist, from the simplest ones that are defined by regular expressions (*e.g.* the above grammar is a typical regular one), to the more complex grammars that can produce any Turing-complete languages. As proposed by N. Chomsky [38], these grammars can be partitioned in classes or groups, following their capacity of capturing key properties of computer-related languages. He identified four main classes of grammar and proposed a well-known framework, the Chomsky Hierarchy illustrated on figure 2.10, to classify them following their expressive power. Each grammatical class (type-0, type-1, *etc.*) denotes both a typical kind of language (regular, context-free, *etc.*) but also a specific tool, or set of tools, that can be used to represent it (finite state automaton, Turing machines, *etc.*). In the remainder of this section, we briefly survey these types of grammars since communication protocols make an heavy use of them to specify their valid exchange of symbols.

The most restrictive type of grammar in this hierarchy is the **type-3 grammar**, also called regular grammar. Such grammar describes a regular language and as established by the Kleen theorem [78], can easily be transformed into a finite state automaton. Its production rules must respect the three following constraints:

— the left part of the PR must be a single nonterminal symbol,

— the right part of the PR must be a single terminal symbol possibly followed by a single nonterminal (left-regular) or a single terminal symbol preceded by a single nonterminal (right-regular).

— the language is no more regular if its combines both right-regular and left-regular rules.

The grammar detailed in 2.3 is a typical example of a regular one.

A **type-2 grammar**, also called a Context-Free Grammar (CFG) or an algebraic grammar, defines context-free languages. Such language can be represented by a non-deterministic pushdown automaton and follows production rules of the form $V \to \gamma$ with γ a sequence of terminals and nonterminals symbols. In addition, the rule $S \to \epsilon$ is valid if S does not appear on the right side of any rule (ϵ denotes an empty symbol). A well-known subset of these languages, the set of deterministic context-free languages, is used by most programming languages supporting the notion of declaration scope as demonstrated by Ogden's lemma [101]. Production rules listed in 2.4 produce a typical example of a context-free grammar accepting the following derivation: $S \to AS \to 0A1S \to 0011S \to 0011AS \to 001101$.

```
S → AS
S → ϵ
A → 0A1
A → A1
A → 01
```

Listing 2.4– Example of production rules that describe a context-free grammar

The upper level of expressive grammar regroups **type-1 grammars**, also called Context-Sensitive Grammar (CSG). Such grammar are defined with production rules in which the left part and the right part may be surrounded by terminal and nonterminal symbols. More formelly, CSG production rules are of the form $\alpha V\beta \to \alpha\gamma\beta$ where α and β denotes a potentially empty sequence of terminals or nonterminals symbols and γ a nonempty sequence of terminals and nonterminals symbols. As in a CFG, the rule $S \to \epsilon$ is valid if S does not appear on the right side of any rule. In the field of communication protocols, the recurrent usage of specific type of fields such as format distinguisher fields that identify the format of the subsequent part of the message reflects the context-sensitivity of their grammars and so of many communication protocols [43].

A typical example of such grammar generates a contextual language $L = \{a^n b^n c^n | n \geq 1\}$ with productions rules listed in 2.5.

```
S → aSBC
S → aBC
CB → BC
aB → ab
bB → bb
bC → bc
cC → cc
```

Listing 2.5– Production rules of a context sensitive grammar that accepts $L = \{a^n b^n c^n | n \geq 1\}$.

Finally, **type-0** grammars, also called unrestricted grammars, denotes all the languages accepted by a Turing machine, which means no restriction is expressed over its production rules.

Naturally, an implementation of a protocol based on a type-0 grammar is much more difficult to develop and maintain than an implementation of a type-3. By extension, automated inference of type-3 grammars is also easier to achieve than for a type-0 grammar. This observation is based on the complexity of mathematical tools (finite state machine, non-deterministic automate, Turing machines) and existing algorithms to perform their inference. Thereafter, we present different specification languages used to model protocols and give examples of their application.

2.3 Existing Specification Languages

With the beginning of the standardization process initiated by the OSI and CCITT in late 70's, researchers recognized that formal specifications of communications could be useful to their work. Their interest came from the observation that very most specifications written in natural languages are not effective to define clear, concise and precise models for their protocols [19]. In addition to ambiguities in protocol specifications, models defined in natural language are not very helpful for the automation of certain aspects of protocol development cycle whereas formal protocol specifications can support automatic validation of specifications [133], the creation of implementations [3] but also conformance testing and automated protocol security evaluations [91].

At this time, special groups were created to propose the concept of Formal Description Techniques (FDT) and to apply it on protocol definitions. Originally, these groups proposed three different protocol specification languages respectively called Estelle (Extended Finite State Machine Language) [103], SDL (Specification And Description Language) [122] and LOTOS [69] still used to specify recent protocols [125, 31]. It exists also some semi-formal specifications languages such as Abstract Syntax Notation One (ASN.1) [1] and Augmented Backus-Naur Form (ABNF) [42] that only covers the data structure definition and not their processing.

It is now widely accepted that the success of a system development depends on the quality of its design and so of its specifications. For this task, protocol creators can rely on these specification languages and on various specification tools to design their system. In the sequel, we describe some of the most popular specification languages that can be used to define the vocabulary (Section 2.3.1) and the grammar (Section 2.3.2) of a communication protocol.

2.3.1 Specification Languages for Protocol Vocabulary

Among existing protocol specification languages, some focus on the definition of protocols vocabulary. They can be used to specify words, tokens and fields that compose each symbol. In addition to their definition characteristics, most languages were proposed with a dedicated compilers that can automatically produce messages out of the protocol specification. These compilers often denote a recurrent usage of specific data structures, encoding rules and compression algorithms that highly impact the final message format. Therefore, an effective message format inference strategy should consider these typical message formats. In the following, we therefore give some insights on the most recurrent languages used in public specifications of protocols vocabulary: the basic textual specification, the ASN.1 format with its encoding variants, the ABNF language and on Google's Protocol Buffer (ProtoBuf) specification language [6]. Based on these descriptions, we formulate few hints that could leverage inference strategy when applied on them.

Textual Specification

A common and simple approach, originally retained to specify the format of a symbol, is the textual specification of a protocol. It usually consists in a graphical representation of messages as arrays of fields, indexed on their size, coupled with a textual description of their values. For example, the specification of the UDP protocol [106] illustrated on listing 2.6 describes five fields, four of two bytes each, called "Source Port", "Destination Port", "Length" and "Checksum" and a last field, called "data" that denotes the payload of the protocol.

Such graphical representation of the protocol format is combined with a textual description in natural language that details the definition domain of each field. For example, the UDP specifications includes the following description of the "Checksum" field: "Checksum is the 16-bit one's complement of the one's complement sum of a pseudo header of information from the IP header, the UDP header, and the data, padded with zero octets at the end (if necessary) to make a multiple of two octets."

```
 0      7 8     15 16    23 24    31
+--------+--------+--------+--------+
|     Src. Port    |    Dest. Port    |
+--------+--------+--------+--------+
|     Length       |     Checksum     |
+--------+--------+--------+--------+
|          data octets ...
+---------------- ...
```

Listing 2.6– Example of a textual specification that defines UDP message format.

Such textual specification of a protocol format is very common in oldest documentations of protocols such as IP [108], UDP [106], TCP [109] and ICMP [107]. The field array is effective to describe the format of binary protocols with invariant size fields but is not adequate for text protocols relying on delimiters. Furthermore, the textual description of fields definition domain in natural languages lack of conciseness, of precision and are often ambiguous [19].

From an inference perspective, most messages defined with such textual representation are made of fixed-size fields which are byte-aligned. These fields are often sized according to variable types offered by programming languages (*e.g.* integer, float, double, long). Inferring their syntax therefore requires to split messages according to these common variable type sizes (*e.g.* one byte, two bytes, four bytes). However, a textual specification can also describe variable-sized fields where their size depend on the value of other field. Inferring such format requires to search for length

6. Protocol Buffer: `http://code.google.com/p/protobuf/`

fields. These fields can then be used to split the messages and expose the presence of variable-sized fields.

Abstract Syntax Notation One

Abstract Syntax Notation One [1] better known as ASN.1, is a standard for data structure originally defined in 1984 by the OSI, the International Electrotechnical Commission and the ITU. This standard defines a formal notation for the description of data structures independently from machine-specific encoding issues. Originally part of the CCITT X.409 specifications, ASN.1 has moved to its own standard, X.208, in 1988 due to its wide applicability.

It provides various pre-defined abstract basic types such as booleans, integers and strings along with structures to support the definition of customs types (*e.g.* sequences, list and choices). For example, the listing 2.7 comes from the specifications of the Simple Network Management Protocol (SNMP) [30] that exposes its messages format with ASN.1 formalism.

```
Message ::=
    SEQUENCE {
        version        -- version-1 for this RFC
            INTEGER {
                version-1(0)
            },
        community      -- community name
            OCTET STRING,
        data           -- e.g., PDUs if trivial
            ANY        -- authentication is being used
}
```

Listing 2.7– Definition of an SNMP `Message` (RFC 1157) using ASN.1 notation.

As shown on listing 2.7, an SNMP message is defined as a sequence of three fields: a version number, a string (*i.e.* a binary data whose length is a multiple of eight) that indicates the message community and a payload. This definition is detailed on listing 2.8 with the specification of an SNMP `GetRequest` message that includes integers to represent the request indentifier (request-id), the error-status and the error-index. It also specifies a list of pairs of name (ObjectName) and values (ObjectSyntax) with some custom types declared in the RFC1155-SMI and imported in the ASN.1 specification of the SNMP protocol.

```
IMPORTS
    ObjectName, ObjectSyntax, NetworkAddress, IpAddress, TimeTicks FROM
        RFC1155-SMI
...
PDUs ::=
     CHOICE {
         get-request
             GetRequest-PDU,
         get-next-request
```

```
             GetNextRequest-PDU,
         ...
     }
...
GetRequest-PDU ::=
    [2]
        IMPLICIT PDU
...
PDU ::=
    SEQUENCE {
        request-id
            INTEGER,
        error-status      -- sometimes ignored
            INTEGER {
                noError(0),
                tooBig(1),
                noSuchName(2),
                ...
            },
        error-index       -- sometimes ignored
            INTEGER,
        variable-bindings -- values are sometimes ignored
            VarBindList
    }
...
VarBind ::=
    SEQUENCE {
        name
            ObjectName,
        value
            ObjectSyntax
    }

VarBindList ::=
    SEQUENCE OF
        VarBind
```

Listing 2.8– Specification of the `GetRequest` SNMP message using ASN.1 notation.

In addition to data structures, ASN.1 also provides various encoding rules, referred to *transfer syntax*, to specify the exact sequence of bytes used to encode each data item described with its notation. Among existing encoding rules, six different encoding rules are very common, the Basic Encoding Rules (BER), the Canonical and Distinguished Encoding Rules (CER, DER), the Packed Encoding Rules (PER) and the XML Encoding Rules (XER). In the following, we give a short description of them, the interested reader can refer to their official specifications published on ITU's website for more details [7]

Basic Encoding Rules (BER) is the default transfer syntax used to encode an ASN.1 message defined under ITU's X.690 standard [72]. With this encoding rule, messages follow a Type-Length-Value (TLV) format where each data is represented by its type, its length and its values. A unique identifier specifies the type. The standard establishes the value of this identifier for every basic types such as `0x2C` for an integer or `0x30` for a sequence while custom ones can be declared in the specifications of the protocol. For example, the header of an SNMP message is a sequence made of its version field, an integer set to zero and a community string. Using BER encoding rules, this header is encoded in bytes as illustrated on figure 2.11.

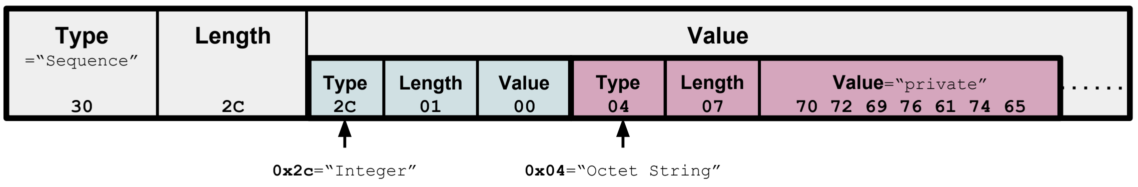


Figure 2.11 – BER Encoding of the SNMP header.

BER and more generally a TLV message format is made of two fixed-size fields (type and length) and one variable-size field (value). This value field can also contain other TLV fields as illustrated on Figure 2.11. Besides, BER may also introduce an optional fourth field called "end-of-contents" right after the value field. This fourth field plays the role of a delimiter for the value field when the BER *indefinite-length* encoding method is used. This encoding method is preferred when large contents are stored in the value field. In such case, a specific value is stored in the length field (`80`) which indicates to the parser that all the remaining bytes belong to the value field until it identifies the value stored in the fourth field.

The reverse engineering of such data structure implies the identification of field boundaries. Given that the size of the value field is variable, an efficient approach could rely on the identification of the length field. As we detail in Section 6.4, such algorithm is more effective if its applies on multiple messages that share the same data structure. This operation could be achieved if we first regroup messages based on the value contained in their type fields.

Canonical and Distinguished Encoding Rules (CER & DER) are restricted variants of BER also described in ITU's X.690 standard. They are both used to produce an unequivocal encoding of a data structure. They differ from BER which gives various choices as how the value field is encoded. For example, BER accepts multiple values to encode the value of the Boolean `TRUE` whereas a single one is allowed in DER. CER and more commonly DER encoding rules were mostly created to encode cryptographic materials such as certificates (PKCS, X.509 certificates, *etc.*). They ensure that a certificate is always encoded with the same byte flow whereas different BER implementations can produce different byte flows for the same certificate. For example, Listing 2.9 illustrates the ASN.1 specification of a X.509 certificate.

7. `http://www.itu.int/en/ITU-T/asn1/Pages/asn1_project.aspx`

```
Certificate ::= SIGNED SEQUENCE{
    version [0]             Version DEFAULT v1988,
    serialNumber            CertificateSerialNumber,
    signature               AlgorithmIdentifier,
    issuer                  Name,
    validity                Validity,
    subject                 Name,
    subjectPublicKeyInfo    SubjectPublicKeyInfo}

    Version ::= INTEGER {v1988(0)}

    CertificateSerialNumber ::= INTEGER

    Validity ::= SEQUENCE{
        notBefore       UTCTime,
        notAfter        UTCTime}

    SubjectPublicKeyInfo ::= SEQUENCE{
        algorithm           AlgorithmIdentifier,
        subjectPublicKey    BIT STRING}


   AlgorithmIdentifier ::= SEQUENCE{
        algorithm       OBJECT IDENTIFIER,
        parameters      ANY DEFINED BY algorithm OPTIONAL}
}
```

Listing 2.9– ASN.1 specifications of X.509 certificates as defined in RFC 1422 [76].

CER and DER differ in the set of constraints they ensure. The basic difference that exists between them is that the former supports the *indefinite-length* encoding method (where a delimiter is used instead of a length field) whereas DER not.

Hopefully, these additional constrains can be helpful to reverse engineer messages that are CER or DER encoded. They ensure that messages of the same type are encoded similarly regarding fields order and value.

Packed Encoding Rules (PER) is a non-TLV compressed transfer syntax, standardized in ITU-T X.691 specifications, that uses a minimum number of bits to encode data. Each data can be specified with its length and its range to optimize the encoding. This way PER is much more compact than BER but requires a decoder that knows the complete abstract syntax. Following the processing capacities of the decoder, PER can also be configured to align encoded values to improve the compression rate. A variant, the CANONICAL-PER, introduces CER/DER-like constraints to support its usage in cryptographic protocols.

The PER encoding format and more generally, any packed format, is much more complex to infer as only variable-sized data often compose them. The message size could be an interesting measure to identify same type messages. Besides, searching for potential embedded environmental information such IP addresses, hostnames and dates could be helpful to identify field boundaries.

XML Encoding Rules (XER) are a set of encoding rules that uses an XML-based representation to encode ASN.1 messages. XML encoded messages are often used in web services and network protocols close to the end-user as being both human and machine-readable. For example, the Common Alerting Protocol (CAP) [134] that allows the exchange of "all-hazard" emergency alerts and public warnings over all kinds of networks, uses a XER transfer syntax. Listings 2.10 and 2.11 illustrates the XER-based specification of an alert and a speculative instance of such alert, both provided in the protocol standard document. Various extensions exist of the XER transfer syntax, including a canonical form, denoted cXER, similar to CER/DER.

```
DEFINITIONS XER INSTRUCTIONS AUTOMATIC TAGS ::=
-- CAP Alert Message (version 1.2)
BEGIN

Alert ::= SEQUENCE {
   identifier IdentifierString,
       -- Unambiguous identification of the message
   sender     String,
       -- The globally unambiguous identification of the sender.
   sent       DateTime (CONSTRAINED BY {/* XML representation of the
      XSD pattern "\d\d\d\d-\d\d-\d\dT\d\d:\d\d:\d\d[-,+]\d\d:\d\d"
      */}),
   status     AlertStatus,
   msgType    AlertMessageType,
   source     String OPTIONAL,
   scope      AlertScope,
   restriction String OPTIONAL,
   addresses   String OPTIONAL,
   code-list   SEQUENCE SIZE((0..MAX)) OF code String,
   note        String OPTIONAL,
   references  String OPTIONAL,
   incidents   String OPTIONAL,
   info-list   SEQUENCE SIZE((0..MAX)) OF info AlertInformation  }
```

Listing 2.10– Specification of a CAP alert message as edited in the ITU-T 1303 recommendation

```
<?xml version = "1.0" encoding = "UTF-8"?>
<alert xmlns = "urn:oasis:names:tc:emergency:cap:1.2">
  <identifier>43b080713727</identifier>
  <sender>hsas@dhs.gov</sender>
  <sent>2003-04-02T14:39:01-05:00</sent>
  <status>Actual</status>
  <msgType>Alert</msgType>
  <scope>Public</scope>
  <info>
    <category>Security</category>
    <event>Homeland Security Advisory System Update</event>
    <urgency>Immediate</urgency>
    <severity>Severe</severity>
    <certainty>Likely</certainty>
    <senderName>U.S. Government, Department of Homeland Security</
       senderName>
    <headline>Homeland Security Sets Code ORANGE</headline>
    <description>The Department of Homeland Security has elevated the
       Homeland Security Advisory System threat level to ORANGE / High
        in response to intelligence which may indicate a heightened
       threat of terrorism.</description>
    <instruction> A High Condition is declared when there is a high
       risk of terrorist attacks. In addition to the Protective
       Measures taken in the previous Threat Conditions, Federal
       departments and agencies should consider agency-specific
       Protective Measures in accordance with their existing plans.</
       instruction>
    <web>http://www.dhs.gov/dhspublic/display?theme=29</web>
    <parameter>
      <valueName>HSAS</valueName>
      <value>ORANGE</value>
    </parameter>
    <resource>
      <resourceDesc>Image file (GIF)</resourceDesc>
      <mimeType>image/gif</mimeType>
      <uri>http://www.dhs.gov/dhspublic/getAdvisoryImage</uri>
    </resource>
    <area>
      <areaDesc>U.S. nationwide and interests worldwide</areaDesc>
    </area>
  </info>
</alert>
```

Listing 2.11– A XER encoded message about a speculative US. Homeland Security Advisory Alert.

Messages XER-encoded are often self descriptive, *i.e.* field names and values can easily be spotted as they are enclosed in XML tags. Indeed, XML encoded messages require no specific inference technique as this format is human-readable.

Several other transfer syntax rules exist but are rarely used, such as the GSER detailed in RFC 3641 [87]. This encoding rule produces a human-readable straightforward textual representation to encode messages with the purpose to display them to the end-user.

To conclude, the ASN.1 is a very common notations in the field of message format specifications. Many protocols use this formalism to specify their message format such as LDAP [146], X.509, Kerberos, SNMP or SIP and a quick search for “ASN.1” term returns more than 506 RFCs [8]. This language is also used in more closed-source protocols such as in air-ground and ground-ground protocols employed by the Federal Aviation Administration and International Civil Aviation Organization encoded in PER [9].

Augmented Backus-Naur Form

Augmented Backus-Naur Form [42] is a formal data structure specification language for the definition of bi-directional communication protocols. Defined under Internet Standard 68 (STD68), the ABNF notation relies on a context-free grammar to specify most IETF standardized protocols.

An ABNF definition is made of rules that uses operators to support, for example, concatenation, alternative and repetition of rules or terminal range of characters. An ABNF rule follows a basic structure: `name = expression`, with `name` the name of the current rule, `expression` the definition of the rule and = the separator between the rule’s name and its definition. For example, on listing 2.12 extracted from the RFC of Internet Message Format protocol [114], the rule named “to” denotes the concatenation of string “To:” with the result of a previous defined rule named “address-list” that defines a list of email addresses and “CRLF” representing a CR character (ASCII value 13) followed immediately by the LF character (ASCII value 10).

```
to  = "To:" address-list CRLF
cc  = "Cc:" address-list CRLF
bcc = "Bcc:" (address-list / [CFWS]) CRLF
```

Listing 2.12– ABNF definition of destination address fields in IMF protocol [113]

Many standards uses this notation to specify their protocol syntax. However, the extensive usage of functional comments in ABNF specifications denotes the difficulty to have precise and complete definitions. For example, in the SIP ABNF specifications (RFC 2543 [58]) the following comment “should be unique for this originating username/host” complete the formal definition of the “sess-id” field. In most cases, such functional comments aims at introducing context-sensitivity in the specifications. Finally, the lack of specific rules that could precisely define the encoding of each data makes it difficult to specify binary protocols using ABNF notations.

From a practical point of view, reverse engineering a protocol specified under an ABNF notation relies on the identification of field delimiters. Indeed, ASCII protocols such as the ones specified under an ABNF notations relies on these delimiters to expose field boundaries. For example, the SIP specification makes an heavy use of spaces and carriage returns to delimit its fields. A naive approach to reverse engineer some ABNF specified messages could therefore rely on the identification of common ASCII delimiters such as “:”, “;” or CRLF in them. However, no convention exist to establish the list of characters or sequence of characters that can delimit fields. For this reason, such approach requires to consider numerous potential delimiters that can lead to false positives. To address this issue, more effective techniques were proposed. Among them,

8. To search for ASN.1 RFCs we used the RFCSearch service: `http://www.rfcsearch.org`
9. ITU-T website: `http://www.itu.int/ITU-T/asn1/uses/`

sequence alignment algorithms are used by most state-of-the-art automatic reverse engineering work [14, 43, 88]. We detail these complex algorithms in Section 3.1.2.

ProtoBuf

Designed in 2001 by Google, Protocol Buffer (ProtoBuf) is a serialization framework that exposes an Interface Description Language (IDL) to specify data structures of protocol messages. This recent specification language was created with the objectives of improving the readability and the easiness of specifying data exchanges. It also reduces the size of each exchanged messages by means of strong compression algorithms [8]. Other recent IDL exist such as Thrift [10] and Avro [11]. They are fairly similar as they were created with the same objectives. In this discussion, we focus on ProtoBuf as we believe it is the more mature. Released under an open-source license and freely available, this language and its associated tools tend to be used in recent web related applications that handle large amount of data. For example, the Apache Hadoop framework [12] supports ProtoBuf message specifications to handle data exchanges between its cluster nodes.

A ProtoBuf definition is made of data structures (called messages) and services described in a proto definition file (.proto). This file can be compiled with a specific tool (called protoc) to generate the code that can be invoked by a sender or recipient of these data structures. Each message is specified with a set of ordered name-value pairs. Each name-value pair denotes a field with its value its type, its name. It accepts basic value types such as integers, floating-points, booleans, strings and raw bytes. Listing 2.13 is an example of a ProtoBuf specification. It specifies the data structure of a minimalist address book and Table 2.2 lists all the native scalar types this IDL accepts. However, messages specifications can also be defined under a hierarchical definition where the type of a message field can be another message. Besides, each field can be set as optional, required or repeated to produce more complex message definitions.

```
package tutorial;

message Person {
  required string name = 1;
  required int32 id = 2;
  optional string email = 3;

  enum PhoneType {
    MOBILE = 0;
    HOME = 1;
    WORK = 2;
  }

  message PhoneNumber {
    required string number = 1;
```

10. Thrift: http://thrift.apache.org/
11. Avro: http://avro.apache.org/
12. Apache Hadoop: http://hadoop.apache.org

```
    optional PhoneType type = 2 [default = HOME];
  }

  repeated PhoneNumber phone = 4;
}

message AddressBook {
  repeated Person person = 1;
}
```

Listing 2.13– Example of a message specification in ProtoBuf

ProtoBuf Type	Notes	C++ Type
double		double
float		float
int32	*Uses variable-length encoding.*	int32
int64	*Uses variable-length encoding.*	int64
uint32	*Uses variable-length encoding.*	uint32
uint64	*Uses variable-length encoding.*	uint64
sint32	*Uses variable-length encoding. Signed int value.*	int32s
sint64	*Uses variable-length encoding. Signed int value.*	int64
fixed32	*Always four bytes.*	uint32
fixed64	*Always eight bytes.*	uint64
sfixed32	*Always four bytes.*	int32
sfixed64	*Always eight bytes.*	int64
bool		boolean
string	*A string must always contain UTF-8 encoded or 7-bit ASCII text.*	string
bytes	*May contain any arbitrary sequence of bytes.*	string

Table 2.2 – Scalar types supported by ProtoBuf as described in the official developer guide

A ProtoBuf message can be serialized in an optimized binary format. This format is similar to the PER transfer syntax rule of the ASN.1 language. It relies on the *Variable-Length Quantity* [12] encoding method (also known as varint) that serializes integers by means of one or more bytes. Originally created for the MIDI file format, this byte-aligned encoding method uses seven bits per bytes with an additional bit to indicate that more bytes must be considered (or not). With this method, the total number of bits in the encoding result depends on the size of the original integer.

To encode a message, ProtoBuf concatenates all the field keys and values into a byte stream. For example, the Figure 2.12 illustrates the byte flow that represents a message made of three fields. The first field contains the string “Netzob”, the second one a decimal (1337) and the third one a repetition of strings "ProtoBuf" and "Thrift". As illustrated, each field is made of at least two sub-fields that contain 1) the field identifier number and its type and 2) the value of the field. The decimal field in our example message illustrates this. Following the type of the field, an additional sub-field is also used. it contains the length of the value field. In our example, the first field and the third field contain such additional length sub-field.

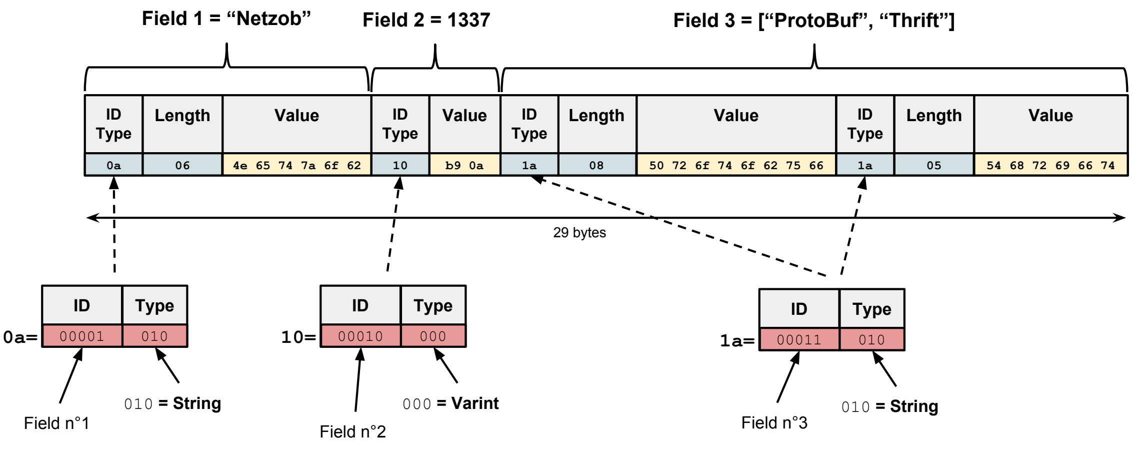


Figure 2.12 – ProtoBuf encoding example of a message.

Automatically reversing a stream byte encoded with ProtoBuf is a complex work. Mostly due to the *varint* encoding method that produces fields of dynamic-sized without any delimiters or explicit length fields. However, not all scalar types are encoded with this method. For example, string values are encoded with traditional UTF-8 alphabet. Besides, as illustrated in our example some length information are sometimes embedded. Reverse engineering an ProtoBuf message could therefore leverage UTF-8 discovering algorithms to first identify potential ASCII fields.

We described in this Section some common specification language that are used by protocol creators to specify the syntax of their messages. We analyzed their specificity and gave some hints on the possible approaches that could be use to reverse them. Most of them rely on the identification of field boundaries through several solutions, *e.g.* the identification of length fields, of ASCII sequences and of environmental information. We also explained the difficulty of reversing compressed messages such as the one generated by a PER encoding function. In the following, we apply the same reasoning and analyze common specification languages that can be used to describe the grammar of a protocol.

2.3.2 Specification Languages for Protocol Grammar

Message Sequence Chart

The Message Sequence Chart (MSC) is an interaction diagram standardized by the ITU [73] related to the languages and general software aspects for telecommunication systems (Z series). This diagram depicts the order in which communications and other events take places between protocol logical processes, their system and their environment. As illustrated on figure 2.13, processes, also called entities or instances, are represented by vertical lines while message exchanges between them are depicted by arrows. Thus, an MSC models communications through message-passing via reliable FIFOs. Its a high-level description of the possible usage scenario but only specifies message orders. The internal behavior of the each process is not considered. Besides, an MSC exhibit a weak partial order semantic that cannot express constraints between message exchanges. For instance, such diagram cannot be used to model that "if $\mathcal{P}$ sends $\mathcal{M}$ to $\mathcal{Q}$, $\mathcal{Q}$ *must* pass on this message to $\mathcal{R}$" [59]. For this reason, such description of a protocol grammar is often limited to capture system requirements in the form of "good" scenarios that the implemented system should exhibit.

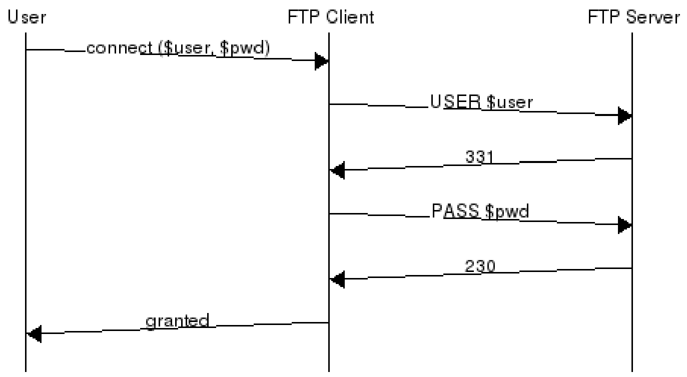


Figure 2.13 – Message Sequence Chart describing a sample FTP authentication process.

Language of Temporal Ordering Specification

Language of Temporal Ordering Specification (LOTOS) is another formal specification language [69] developed within the ISO between 1981 and 1984. The key idea behind the LOTOS specification of a protocol is to describe the temporal relations that exist between observed externals events (from a system point of view). Some key principles have inspired its design, such as:

— A syntactic and semantic separation is ensured between the definition of processes and the definition of types.
— The operational semantic are defined using an algebra approach, mostly inspired by CCS/CSP-based language [95, 64] in such a way that it is possible to prove a rich set of algebraic equivalence properties.

.

A LOTOS specification is an ASCII text that describes a set of processes and type definitions. A *process* is a black box abstraction of an activity in an implementation, for which only its external behavior is considered. Processes are synchronized using a relative temporal ordering of events and share communication mechanisms called *interaction points*. It supports the description of data and operations based on abstract data types, a mathematical model for similar data structures.

The interested reader can refer to the LOTOS introduction [90] by L. Logrippo *et al.* which gives a complete definition of all the concepts behind this protocol specification language. This FDT has been widely used for defining common OSI protocols in academic works [93]. In practical, protocol development LOTOS has attained little relevance [80].

Estelle

Published in 1989 [103], Estelle is an ISO standard specification language, capable of defining concurrent and distributed communication protocols. Based on a formal definition, it aims at identifying and mitigating any possible ambiguities in protocol implementations.

To achieve this, an Estelle specification relies on two parts, 1) the architecture and 2) the behavior. The architecture defines a hierarchy of various modules, or actors of a communication while the behavior denotes how actors handle messages based on a finite state machine with memory, *i.e.* an Extended Finite State Machine (EFSM). It models a system as a hierarchy of structures that can run in parallel, exchange messages and share some variables. As illustrated on figure 2.14, two modules interacts through a channel interconnected on their interaction points.

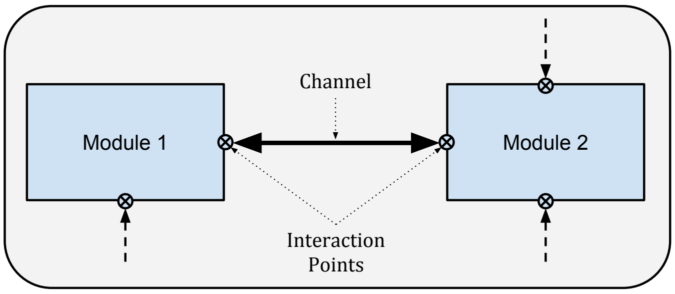


Figure 2.14 – Sample Estelle architecture.

To model interactions between modules, exchanged messages are stored in FIFO queues that enable the use of conditional transitions in the EFSM, *i.e.* a transition is fired when all enabled conditions are fulfilled. Additional rules can also be used to specify synchronous and asynchronous transition properties.

Specification and Description Language (SDL)

Defined by the International Telecommunication Union (ITU) in 1992, the SDL formal language is intended for the specification of reactive, real-time, and distributed applications involving many concurrent activities. Very most of communication protocols can therefore be described with such language. For example, it exists some SDL specifications for the LTE and DSR protocols [125, 31]. It allows to specify the functional properties of the system and their relationships with the environment.

A graphical representation (SDL/GR) and a textual representation (SDL/PR) are proposed to describe the structure, the behavior and the data of a protocol. The graphic form is preferred for most people as shown by its usage in most academic papers. The interested reader can refer to the reference book on SDL [47].

All these models that can be use to specify the grammar of a communication protocol are complex. Their rely on mathematical tools such as EFSM that are extended with different controls to ensure their large coverage of protocol requirements. These specification languages can be use to model probabilistic and distributed protocols. We believe such models are far too complex to be inferred with existing grammar inference algorithms. We therefore focused our work on learning deterministic mealy machines.

Chapter 3

Communication Protocol Inference

This chapter exposes previous works in the field of the automated inference of a communication protocol. Section 3.1 reviews the different approaches in the field of vocabulary inference while Section 3.2 covers previous work in the field of grammatical inference applied to the RE of protocol grammar.

3.1 Automated Inference of the Vocabulary

As described in section 2.2, a protocol is made of a vocabulary that defines the set of accepted messages with their definition and a grammar denoting the set of accepted sequences of messages. Thus, an inference process must address both to properly reverse an unknown protocol. However, the grammatical inference of a protocol requires some previous knowledge over the vocabulary. For this reason, the reverse engineering of a protocol traditionally starts with the vocabulary inference.

Previous work in the field of automated inference of the vocabulary falls into two families depending on whether they analyze an implementation of the protocol [27, 29, 41] or rather some communication samples [88, 14, 43, 83, 139, 138, 82].

Works that participate in the first family analyzes the executable binary that implements the targeted protocol. They observe the parsing process for received messages and the buffer construction method for sent messages. Results brought by these works seemed to be efficient to retrieve the compositional nature of messages in fields. However, they suppose the use of static analysis and intrusive dynamic techniques on binaries. We believe this approach cannot be easily automated, mostly due to its complexity but also because of existing counter-measures such as static and dynamic obfuscation, code compression, anti-debugging and anti-instrumentation solutions.

Therefore, we focused our work on the second family of vocabulary reverse engineering approaches. Contrary to the first ones, this family of trace-based vocabulary inference approaches only rely on collected messages to infer the vocabulary of an unknown protocol. Messages can be extracted out of a captured communication trace, for instance from a pcap file for network protocols. We believe this approach brings fewer assumptions over the targeted protocol and its implementation and for this reason is more practical. Nonetheless, trace-based approaches are more sensible to encryption than binary-based approaches as they rely on pattern matching algorithms that are not effective on encrypted messages. However, solutions exist that could be use to tackle this encryption issue [28, 140, 4, 26]. Some of them imply a partial reverse engineering of the implementation to collect exchanged messages before their encryption [28, 140]. For example, specific probes can be use to extract unencrypted sent and received messages that are hosted in some buffers of the program. Such operation is easier than the complete reverse engineering of the protocol implementation. Besides, we do not consider these two families as completely orthogonal and future works could combine our methodology with results brought by a binary analysis.

Among all the existing issues encountered when inferring the vocabulary of a protocol using such trace-based approach, we retained generic ones either clearly identified and addressed by state of the art work or that we faced while building our own trace-based inference solution. Thus we highlight three recurrent issues: 1) message extraction, 2) identification of equivalent messages and of their format and 3) relationship inference. The first common issue is related to the identification, in provided traces, of message boundaries. We detail existing work to address this issue in section 3.1.1. The second issue, detailed in section 3.1.2 comes from the difficulty of identifying equivalent messages and their format in a set of collected traces. Finally, works that identify and infer field relationships, such as size fields and sequence numbers are detailed in section 3.1.3.

3.1.1 Extracting Messages from Traces

As stated below, a trace-based vocabulary inference relies on collected communication traces to infer the vocabulary of an unknown protocol. By traces, we refer to detailed records of communications between two actors that includes exchanged byte flows labeled with their direction and their timestamps, *e.g.* actor A sent `0xA1A2A3` to actor B the $11th$ of May 2014 at 12:24:26 UTC. A Sequence of Events Recorder (SER) such as Wireshark [1] can be used to collect these traces.

The first step in such trace-based approach consists in identifying messages related to the targeted protocol among collected traces. However, the layered architecture used in computer-related communications brings at least two issues. The first one is brought by the presence, in traces, of bytes related to sub and upper layered protocols that can prevent the inference process to effectively apply on the targeted protocol. The second one regards the fact that each layer may have a specific fragmentation and concatenation strategy that can prevent the correct identification of message boundaries in provided bytes flows. In this section, we detail these issues and present how state of the art solutions propose to address them.

Filtering Unrelated Protocols from Traces

As explained in Section 1.1, protocols are often designed using a layered-based approach. For this reason, a communication trace often includes multiple protocols, one for each layer. Typically, each layer *prepend* and/or *append* additional information to messages passed down from upper layers. At each layer, a message extended with additional information forms the Protocol Data Unit

1. Wireshark is a famous free network capture tool: `http://www.wireshark.com`

(PDU) of the protocol. As illustrated in Figure 3.1, PDUs are recursively encapsulated one into the other.

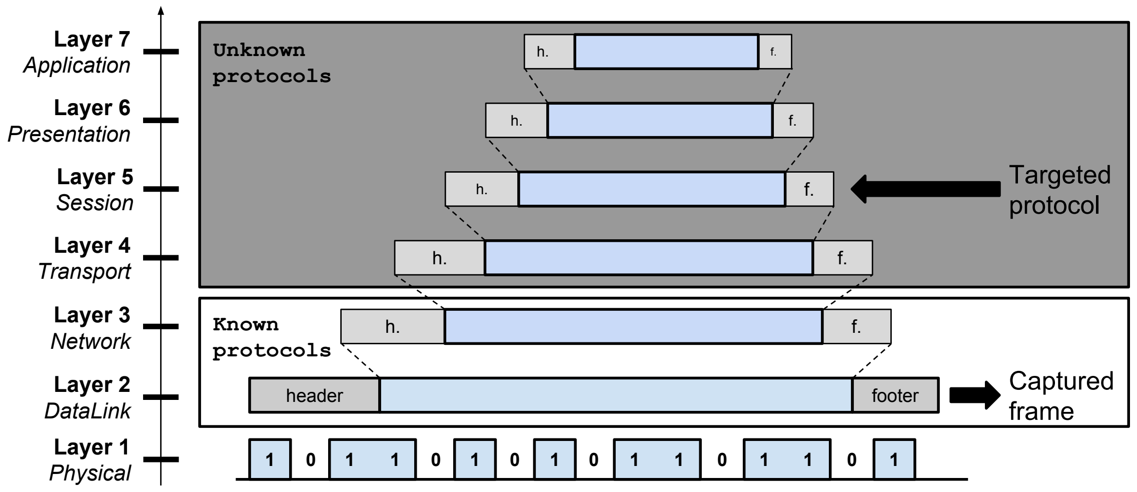


Figure 3.1 – Protocol layering

The recursive encapsulation of protocols makes difficult the identification of PDUs related to the targeted protocol. It implies to filter out headers and footers generated by underneath protocols. To filter these, most work in the field of vocabulary inference [14, 139, 88] rely on sufficient knowledge over beneath protocols to isolate and remove them. This approach supposes that protocols are reversed under a specific order: from the lowest layers to the upper layers. For example, the **PI** tool proposed by M. Bedoe [14] only applies on traces that contains HTTP messages removed from any content introduced by underneath protocols such as TCP, IP and ethernet. **ScriptGen** [88] and **Veritas** [139] are other protocol reverse engineering tools that follow this approach.

Identification of Message Boundaries in the Traces

As described previously, messages are organized in a layered hierarchy. Each layer provides a set of rules that govern communications between systems. Despite the ones that establish the vocabulary and the grammar, a protocol also implements a flow management strategy. This strategy describes how a connection is initiated, maintained and closed but also how messages are exchanged. Among other things, this strategy aims at reducing the processing effort and adapt message exchanges to the communication channel. To achieve this, such protocol can fragment or aggregate messages that are generated by their above layers. These modifications must be considered when collecting traces. In the remainder, we focus on the modifications introduced by both the stream oriented protocols such as TCP and protocols that fragment messages such as IP protocol to illustrate two common modifications of messages.

Stream oriented protocols such as TCP tend to hide all the specifics of the underlying protocols to its above layers and among them the notion of packets. Instead, such protocol introduces the notion of stream that represents a continuous flow of sent received bytes. It exposes this stream to its upper layer protocols. Applications successively read and write a specific amount of data on this stream. However, a TCP stream provides no information that could be use to delimit successive sent or received messages. Thus, if an application successively sends two messages, these messages can appear as a single one to an observer that does not know the protocol specification. For example, in the Figure 3.2, an application sends four successive messages. As they reach the fourth layer, these messages are aggregated into a stream without any information that could be use to identify their boundaries. For this reason, reverse engineering a TCP based protocols by means of collected streams is not straightforward. It requires to identify message boundaries in the stream. A solution retained by most works [88, 43] relies on the assumption that a message can be defined as the longest consecutive set of bytes going in the same direction. This heuristic cannot be use to identify the boundaries of successive messages going in the same directions. Some other solutions [43] leverages Wireshark to parse captured messages and identify boundaries prior to their inference. However, it requires prior-knowledge over the targeted protocol to be effective.

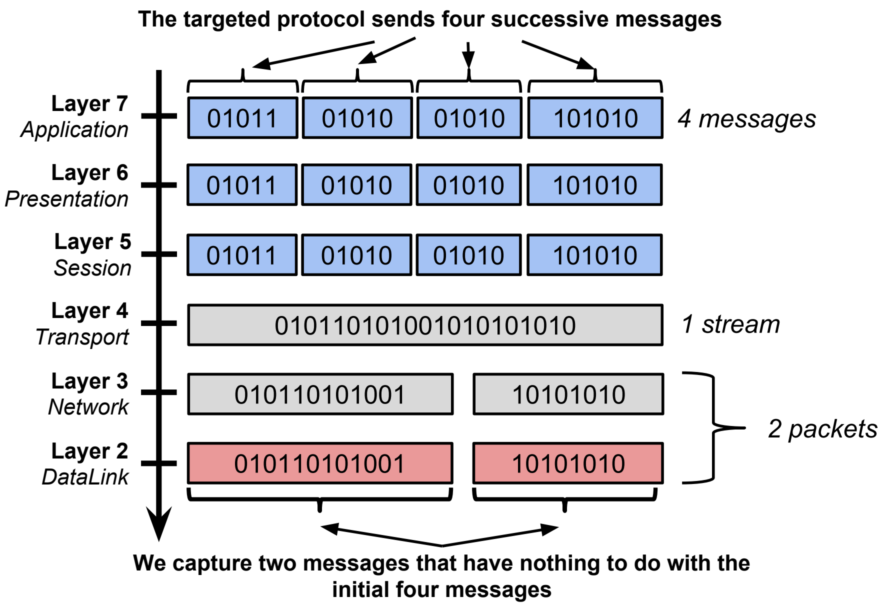


Figure 3.2 – Protocol layering

The fragmentation is another common modification of messages that impacts trace-based reverse engineering works. Indeed, as specified in the OSI model, each layer uses the service exposed by its underneath layer. However, a layer is not always aware of the largest size of message the underneath protocol supports. The IP protocol refers to the Maximum Transmission Unit (MTU) to designate the largest size of a message it can transfer. Depending on the network architecture this MTU may change (*e.g.* LAN and MAN networks tend to accept larger MTU than WAN networks). Indeed, messages generated by above layers sometimes exceed the that rely on IP MTU of the IP layer. In such case, the IP protocol fragments the message into smaller message chunks (less than the MTU) in order to allow it to be received by the final destination system. For example, Figure 3.2 illustrates how a TCP stream can be fragmented into two IP packets. It impacts the reverse engineering of IP-based protocols as messages may be captured in multiple chunks. These chunks must be reassembled to obtain the original message. This operation is complex as chunks may be received out of orders and since nothing prevent the creation of overlapping chunks. For this reason, previous knowledge over the IP protocol is required to reassemble received messages. Furthermore, IP is not the only protocol that implements such restriction over its message size. For example, TCP protocol also includes a restriction on its segment size called the Maximum Segment Size (MSS). A similar restriction is employed by the HTTP protocol that supports the fragmentation of its content in chunks. To our knowledge, the only retained approach to address this issue is to reverse engineer protocols that rely on known ones.

3.1.2 Identification of Equivalent Messages and Inference of their Format

As explained in Section 2.2, the vocabulary of a protocol can be modeled under a symbolic form. As a remainder, a symbol abstracts equivalent messages from a protocol perspective. By equivalent message, we refer to messages that have the same semantic definition from a protocol perspective and share the same message format. Therefore, an important step in the inference process of the vocabulary is to identify equivalent messages.

The different approaches that exist to identify equivalent messages are strongly tied to the algorithm they use to partition each message in fields. This observation mostly comes from the assumption made by researchers that two messages that share the same format can be seen as equivalent from a protocol perspective. Therefore, we regrouped both these two issues in our comparison of vocabulary inference approaches. We identified three different strategies: 1) a token-value clustering that filters out low and high frequency values when comparing messages [83, 139], 2) the use of a token-type clustering algorithm to regroup similar messages following the type of their nested values [43] and 3) an alignment-based clustering that groups messages that share a common alignment [14, 88]. We detail these approaches in the following.

Token-value Clustering

This approach relies on the assumption that equivalent messages share a set of common representative keywords. For example, HTTP request messages can be clustered following the value of their method field that either contain the keyword *OPTIONS*, *GET*, *HEAD*, *POST*, *PUT*, *DELETE*, *TRACE* or *CONNECT*. To identify these keywords, a tokenization algorithm often based on n-grams to split messages in tokens is used. As a remainder, we explained in Section 2.2.1 that a token is sequence of bytes that share a same meaning. Then, these approaches applies statistical tools such as Kolmogorov-Smirnov (K-S) Test Filers [33] or Non-Negative Matrix Factorization [86] to identify the most representative tokens or keywords. In the sequel, we present two state-of-the-art works that reverse a protocol vocabulary by means of a Token-Value Clustering.

Veritas published by W. Yipeng [139] relies on this approach to cluster equivalent messages. At first, to identify tokens it splits the first twelve bytes of a each message in 3-grams. Then authors identify the most frequent tokens by means of a Kolmogorov-Smirnov Test filter. Retained tokens are called *candidate message units* and can be seen as keywords. In a second step, the tool analyzes the frequency of each keyword in the collected messages. If the frequency exceeds a threshold, the candidate message unit represents a cluster. Thus every messages that embed this keyword are said equivalent and clustered together.

Another work that leverages a token-value clustering approach is **ASAP**, published by T. Krueger *et al.* [83]. Similarly to **Veritas**, **ASAP** splits messages in n-grams and searches for the most representative ones. To achieve this, they filter out n-grams that have extreme (high and low) frequency of appearance. Indeed, they assume that both constant and highly volatile n-grams do not augment semantics. A Non-Negative Matrix Factorization [86] is then performed to cluster similar messages based on their n-grams. Finally, a template is built for each cluster with the succession of n-grams it contains to represent its associated message format.

As shown in our comparative study detailed in Chapter 7, this approach produces invalid clusters as different types of messages tends to be regrouped together. This mostly comes from the wrong assumption that one of more keywords are enough to split messages in clusters. In practical, protocols may include values that could appear as keywords without denoting a specific type of message. Finally, message formats produced by both this tools are coarse-grained and not precise enough as they solely rely on n-grams. Our study shows that produced message format are not effective to parse protocol messages neither to generate new valid messages.

Token-type Clustering

The token-type clustering approach differs from the previous one by considering the type of embedded tokens instead of their values when clustering messages. **Discoverer** by W. Cui *et al.* [43] is a typical example of a tool that uses this approach. Its tokenization process splits messages in text and binary tokens. A text token is made of any successive sequence of two or more ASCII value bytes that contains no text delimiters (*i.e.* space or tab). A binary token represents a single byte that is not part of an text token. Messages that share the same sequence of tokens and that were sent in the same direction, *i.e.* received or sent, are clustered together. This token-type clustering is illustrated in Figure 3.3.

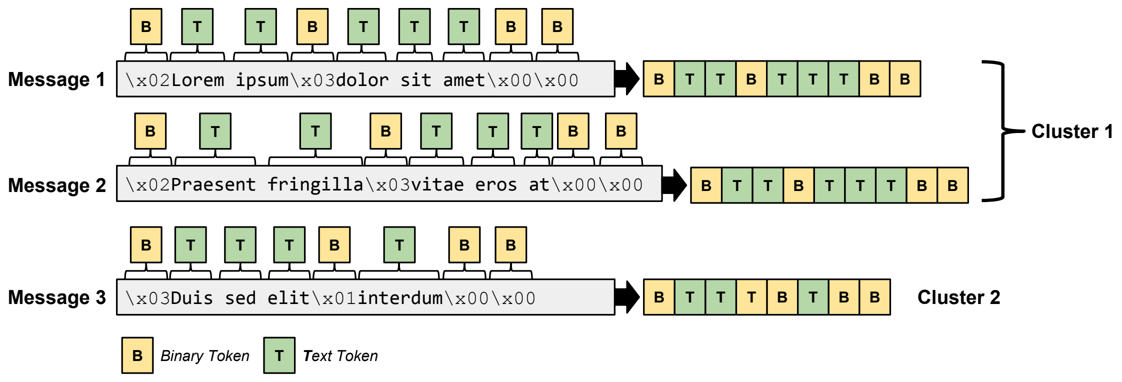


Figure 3.3 – Clustering message based on binary and text tokens

Discoverer also implements a token-value clustering algorithm to subdivide the obtained clusters by identifying "format distinguisher" tokens among them. A format distinguisher token is either a text or a binary token that satisfies two main criteria. First, the number of unique values taken by this token across the set of messages is less than a predefined threshold (for example, ten unique values). In our example illustrated in Figure 3.3, the second token of the first cluster accepts two values: `Lorem` and `Praesent`. If the first criteria is satisfied, a second criteria ensures that it exist at least one value accepted by this token that is present in a minimum number of messages. This additional threshold is referred to as the minimum cluster size and is arbitrary set to 20 messages. If both criteria are satisfied, messages are sub-clustered according to the value of this token and this algorithm is repeated on each sub-cluster.

To mitigate over-classification problems, a last step performed by **Discoverer** approach merges similar message formats by means of a type-based sequence alignment. We describe this clustering approach in the next Section.

Alignment-based Clustering

The alignment-based clustering approach relies on an alignment algorithm such as Needleman & Wunsch (NW) [97] to compute the optimal alignment between two messages. The quality of this alignment is then estimated though different measures such as the number of bytes that perfectly match and the number of mismatches. A similarity score is then produced out of these measures and is latter used by a clustering algorithm to regroup message that best align together. The Unweighted Pair Group Method with Arithmetic mean (UPGMA) algorithm [127] is an example of such clustering algorithm.

The work of A. Beddoe [14] that led to the creation of **Protocol Informatic project (PI)** [2] is among the first to propose this alignment-based clustering approach. To achieve this, **PI** leverages the UPGMA clustering algorithm to organize messages in a phenetic tree. Based on a pairwise similarity matrix filled with the results of messages alignment, this algorithm computes a score between group of messages that reflects their similarity. It then recursively merges the nearest ones until the score drops to a predefined threshold. In the initial state, a cluster is created for each message. Instead of recomputing the similarity matrix when two clusters are merged, an approximation is used. The distance between two clusters A and B denoted $d(A, B)$ is taken to be the average of all the similarity score ($D_{x,y}$) between pairs of x in A and y in B as illustrated by equation 3.1.

$$d(A, B) = \frac{1}{|A| \times |B|} \sum_{x \in A} \sum_{y \in B} D_{x,y} \tag{3.1}$$

In this approach, the initial similarity matrix is filled with a score ($D_{x,y}$) that denotes the quality of the alignment of message x with message y is computed with NW algorithm. This algorithm, also known as the optimal matching algorithm, uses dynamic programming to align two messages.

In NW, the alignment of two messages m_1 and m_2 is performed in two steps. First, a matrix F including a column for each byte of m_1 and a row for each byte of m_2, is created. We denote $m[x]$ the byte of index x in message m. This matrix is then filled accordingly to the principle of optimality described by formula 6.1. It uses a gap penalty d and a similarity function S.

$$F_{i,j} = max(F_{i-1,j-1} + S(m_1[i], m_2[j]), F_{i,j-1} + d, F_{i-1,j} + d) \tag{3.2}$$

2. PI: `http://www.4tphi.net/~awalters/PI/PI.html`

In previous works [14, 43, 88], the similarity function S is reduced to a simple function $v(a,b)$ returning a match or mismatch coefficient, respectively e and f:

$$S(a,b) = v(a,b) = \begin{cases} e, & \text{if } a = b \\ f, & \text{otherwise} \end{cases} \tag{3.3}$$

For example, Table 3.1 illustrates the NW matrix built to align two binary messages: m_1 =`70832F65BD867AD200` and m_2 =`70C400D200` with a match coefficient ($e = 1$), a mismatch coefficient ($f = 0$) and a gap penalty ($d = 0$). We retained the coefficient values proposed by M. Bedoe. Once completed, the similarity score of the two messages can be found at the bottom right of the matrix ($D_{m_1,m_2} = 4$).

		70	**83**	**2F**	**65**	**BD**	**86**	**7A**	**D2**	**00**
	0	0	0	0	0	0	0	0	0	0
70	0	1	1	1	1	1	1	1	1	1
83	0	1	2	2	2	2	2	2	2	2
00	0	1	2	2	2	2	2	2	2	3
D2	0	1	2	2	2	2	2	2	3	3
00	0	1	2	2	2	2	2	2	2	**4**

Table 3.1 – Similarity function $S(a,b)$.

Based on this matrix, the best alignment of two messages can be computed by means of a back-trace step. This step searches for a path that starts at the cell with the highest score, *i.e.* at the bottom right of the matrix, and that maximizes the alignment score back to the origin. As explained by M. Bedoe, this path is constructed after accessing the left, diagonal and upper cell and moves to the one with the maximum score. If all cells are equals, we move to the diagonal. In Table 3.1, yellow cells highlight the computed path. This path can then be used to build a consensus message format that accepts the two messages. This consensus message format is made of static and dynamic fields. To obtain it, **PI** compares the pair of message bytes that are identified by the back-trace step. If their values equals, it adds a static token with this value in the consensus message, if not, it inserts a dynamic token in the consensus. Finally, successive static or dynamic tokens are merged under the same static or dynamic field. Applied to our example, it computes the consensus message format illustrated on Figure 3.4 that is made of three fields.

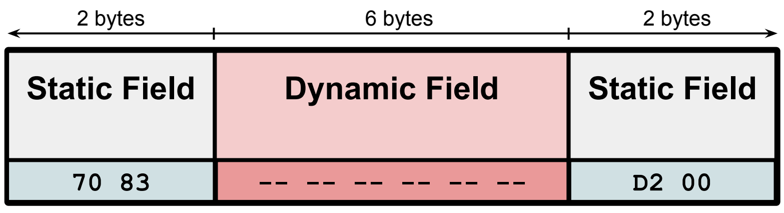


Figure 3.4 – Example of a consensus message format

As shown by M. Bedoe [14], the alignment-based clustering algorithm can be used to identify different messages that share an equivalent format. However, applied to complex protocols, this approach suffers from limitations. The main limitation comes from the assumption that messages are equivalent if their format are equivalent. However, in some cases the type of a message depends of its usages from a protocol perspective. For example, two messages that share the same format may denote a different semantic if one is triggered by the client and the other by the server.

To address these issues, some works [88, 43] propose to pre-cluster messages before executing the alignment-based clustering. This way, only messages that share certain properties are aligned together thus reducing the risk of aligning unrelated messages. For example, we have shown in the previous Section that **Discoverer** relies on a Type-based pre-clustering step to cluster messages that share the same structure of token types (text or binary). In addition to this solution, authors of **Discoverer** proposes a modification of the NW algorithm to align messages based on the type of their tokens instead of their values. We detail both works in the following.

ScriptGen developed by C. Leita *et al.* [88] was initially designed to generate honeypot scripts for unknown protocols. Before executing this alignment-based clustering, it relies on a passively built Finite State Machine (FSM) to execute a pre-clustering step. This FSM denotes the sequence of sent and received message in all the captured sessions of the unknown protocol. Messages that appear in the same state of the FSM are clustered together. Clusters are then subdivided following two heuristics: 1) the number of bytes sent in response to a message and 2) the result of Region Analysis algorithm execution. This last algorithm is applied in two steps, first it clusters messages following an alignment-based clustering algorithm, then it subdivides obtained clusters following messages values.

We already described the approach followed by **Discoverer** to reverses unknown protocols by means of tokenization and recursive clustering. However, to mitigate over-classification problems brought by its recursive clustering algorithm, it introduces a merging step. This last step is similar to the one proposed in **PI** and **ScriptGen** excepts that it relies on the alignment of token types (ASCII or binary) instead of token values. To achieve this, the authors modified the NW algorithm to ensure that two bytes of the same types are aligned while bytes of different types are not.

In this Section, we described different approaches and tools that exist to identify equivalent messages and infer their formats. To the best of our knowledge, no previous work has proposed a comparative study that could permit to identify the solution that best applies on network protocols. In this thesis, we tackled this and compared the results brought by these approaches on various text and binary protocols. Indeed, our comparative study detailed in Chapter 7 shows that the approach followed by **Discoverer** is the best solution of our comparative study. It results outdo other works in the field of automated vocabulary inference when applied on both text and binary protocols. Its combination of multiple steps of clustering with token-type partitioning techniques allows to get the most of each algorithm. However its approach only relies on syntactic comparisons, *i.e.* value and type (text or binary), to delimit the static and dynamic parts of the common format of two messages. It prevents from dissociating two consecutive static (or dynamic) fields, which have different semantics but share the same type. For instance, in DHCP messages, the Server IP Address (`SIAddr`) and the Gateway IP Address (`GIAddr`) are stored in consecutive fields that have the same type (binary) but different meanings. Consequently, these two fields must be separated, which is not possible using syntactic approaches.

To address these issues, inherent to syntactic approaches, we propose an inference algorithm that takes into account the semantic associated with message parts. In particular, our approach does not limit the semantic definition of a message part neither to its data type (text or binary) [43] nor to its appearance probability [138] but instead leverages contextual and environmental information to improve both message clustering and message format inference.

3.1.3 Detecting Field Relationships

To be complete, a format extraction also requires to infer relationships between fields, as they are common in protocols and participate in fields definition. Among all the protocols we observed, the following types of relationships are recurrent: size fields, offset fields and relationships between field values.

A size field specifies the size in bytes of one or more other fields. For example, the IP protocol includes a “total length” field that contains the total length of the packet in bytes. Offset fields are similar as their denotes the position of another field in the message. For example, the IP protocol includes an “Fragment Offset” that specified the offset a particular fragment relative to the beginning of the original unfragmented IP packet. Finally, another common relationship that exists between fields are error-detecting codes such as CRCs or checksums. Such field contains a value that depends on the value of one or more other fields. This value is often the result of a mathematical operation.

The identification of field relationships has been the subject of very few works [43, 88]. Indeed, the computational complexity of identifying relationships between fields often limit existing approaches to the detection of simplest size and offset fields [43]. For example, **ScriptGen** only searches for fields that share the same value. As another example, **Discoverer** relies on an intuition to identify size and offset fields. It assumes that for a specific pair of messages, the difference in the values of potential size fields reflects the difference of the sizes of the messages or some subsequent tokens. It therefore simply check for a match between the value difference and the size difference. If a match holds for all pairs of messages in a cluster, the potential size field is confirmed. For offset fields, they compare the value difference with the difference of the offsets of some subsequent tokens. Both work cannot identify more complex relationships such as message digests or error-detection codes.

From the best of our knowledge, no work has addressed the computational complexity behind the inference of non-linear relationships between fields. Finally, as relationships between fields are also elements of semantic we believe they should be considered during further sequence alignment and message clustering in order to improve their efficiency.

In this analysis of the state-of-the-art, we described recurrent issues previous work have to address in order to reverse the vocabulary of a protocol. We underlined the need in traces that are obtained in a controlled environment that permits to filter out unrelated messages and to reassemble the captured ones. We also detailed the three main approaches that are used by previous works to identify equivalent messages and retrieve their format: token-value clustering, token-type clustering and alignment-based clustering. We described the benefits and the limitations of each one. Based on this analysis and on our comparative study we latter describe in Chapter 7, we can conclude that best results are obtained when a combination of these approaches is used. We rely on this observation for our solution. Besides, we propose a novel approach that takes into account the semantic associated with message parts to improve the identification of equivalent messages and their composition in fields. Finally, we observed the lack of effective solution to infer relationships between fields. In this thesis, we tackle this issue and propose a solution that can identify common linear and non-linear field relationships. We also leverage this information in our clustering and message format inference.

3.2 Automated Inference of the Grammar

We discussed in Section 3.1 the state-of-art in the field of protocol vocabulary inference. We now focus on the second problem, *i.e.* improving the grammatical inference of undocumented protocols. In this Section, we therefore consider solutions published in previous works to infer the grammar of a protocol. In fact, protocol vocabulary inference has been adressed by a larger set of works [88, 14, 43, 83, 139, 138, 82, 27, 29, 41] than for the grammar inference of communication protocols [41, 88, 67, 20, 2, 17]. Indeed, despites the fruitful research field of grammatical inference, few studies have applied existing approaches on communication protocols. The reader interested in the field of grammatical inference can consult the reference book of Colin de la Higuera [63].

The term *grammatical inference* describes all the techniques used to infer a grammatical formalism out of partial information on a targeted language. In general, the inference process uses a "student" that is given access to some data. The student extracts informed samples from them. These samples are made of two sets of labeled strings: *positive examples* (S_+) and *negative examples* (S_-). Such strings can take multiple forms, such as lists of system calls [41], data taken from the Control Flow Graph of an application [124] or network packets [139]. In the following, we focus on the grammatical inference of network communication protocols which therefore relies on samples made of network packets. Nevertheless, our approach also applies in different contexts such as Inter-Process Communication (IPC) or USB protocols. Indeed, the tool we developed as part of this thesis accepts various sample forms besides network packets such as files, IPCes or raw data. Our inferring process only relies on *non-conflicting* samples such that $S_+ \cap S_- = \varnothing$. These samples give insights about the elements of the language used in the protocol and on the rules that explain their sequences. Students aim at inferring a grammar that best justifies the analyzed data.

There are two types of students. The first one analyses communications without participating [9, 139, 124, 41, 67] while the second one takes part and even sometimes initiates the communication [20, 35]. Depending on the type of student used, the inference is named passive or offline for the former and active or online for the latter. The remainder is organized as follows, we describe existing work that rely on a passive grammatical inference in Section 3.2.1 and then focus on state of the art work relying on an active inference process in Section 3.2.2.

3.2.1 Passive Grammatical Inference

In passive inference, the student is fully dependent on observed actors since by definition it does not participate in the communications. Instead, it solely infers the grammar out of provided samples. Some previous work in the specific field of protocol reverse engineering use passive approaches [9, 139, 124, 41, 67]. From a practical point of view, this solution suffers from fewer constraints than the active approach, as no participation in a communication is required. However, given only positive samples S_+ (or $|S_+| >> |S_-|$), inferring the protocol grammar is a hard problem because it implies learning without counter-examples that could help to refrain from generalizing [63]. Indeed, Gold [55] has shown that inferring simple protocols (*i.e.* that can be described by regular languages) requires both positive and negative samples to build an exhaustive model of the targeted grammar. The main problem is that finding negative samples in real life networks is difficult. Such negative samples are often more the result of fuzzing techniques applied on the implementation of the protocol. Thus, unless the provided captures include identifiable bad usages of the protocol, two solutions are available for passive inference approaches: 1) to restrict the language to specific classes of formal languages that has been proven to be learnable from positive samples only [135] or 2) to rely on various heuristics to limit the over-generalization problem [41, 88, 67]. To our knowledge, previous work in the field of protocol inference have only chosen the second solution, the first one being too restrictive.

More specifically, most of previous work rely on the construction of a Prefix Tree Acceptor (PTA) [41, 88, 67] which is a tree-like Deterministic Finite Automaton (DFA). The root of the tree is the initial state of the DFA and each branch represents an application session, *i.e.* a sequence of protocol messages exchanged between a connection and a disconnection. As an example, figure 3.5 illustrates a PTA built with three different application sessions made of positive samples, $S_+ = \{(\texttt{Login}, \texttt{Exit}), (\texttt{Login}, \texttt{Download}, \texttt{Upload}, \texttt{Exit}), (\texttt{Login}, \texttt{Download}, \texttt{Exit})\}$.

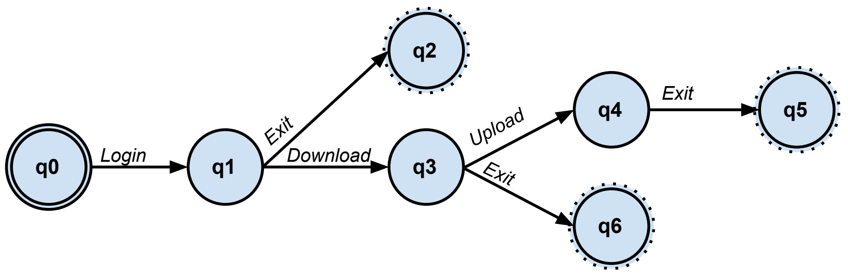


Figure 3.5 – PTA([(`Login`, `Exit`), (`Login`, `Download`, `Upload`, `Exit`), (`Login`, `Download`, `Exit`)])

A DFA minimization algorithm is then used to transform the PTA into an equivalent but smaller DFA, *i.e.* which recognizes the same regular languages with fewer states. This step is the key to most algorithms that deal with inferring an automata out of samples. Various solutions have been proposed but they have in common to generate incomplete models. Indeed, their minimization process relies on an approximation algorithm to support incompletely specified FSM [105]. For instance, T. Xie [144] uses a *k-tail* algorithm that merges states from which possible transitions generate the same future messages (up to an established horizon), Prospex uses an extension of the Exbar algorithm [85] and Hsu *et al.* [67] propose their own offline state merging algorithm to find consistent DFAs out of a built PTA. All these approaches suffer from a scalability issue when applied on large automaton due to the NP-completeness of such algorithms. Our approach only uses passive inference as an initiating step to the active inference process. This way, we address both the completeness and complexity issues by using an active inference.

3.2.2 Active Grammatical Inference

In active inference, the student has access to an "oracle" to which he submits queries. A query is made of messages, each belonging to the protocol vocabulary and denoted $m \in \Sigma$, with Σ, the set of all the messages of the protocol. An oracle is an abstract machine that answers queries about the target. A key algorithm in active inference is the L^* algorithm proposed by D. Angluin [7] which infers DFAs using Membership Queries (MQ) and Equivalency Queries (EQ). L^* originally applies on Moore machines, however in the following we use its adaptation by Niese [98] that can be use to infer Mealy machines. In the following, L^* refers to the Niese adaptation of L^* and applies on Mealy machines.

The approach taken by L^* consists first in building an hypothesis automata $(\texttt{HYP})$ out of numerous membership queries. Formulating a membership query consists in submitting a specific string $u \in (\Sigma_I')^*$ to the oracle and observing the response $\lambda_{\texttt{SUL}}(u)$. Results brought by these membership queries are stored in an observation table. When this table is closed and consistent [7], two properties we detail latter, we build from it an hypothesis automata. Then, an equivalence query is used to verify that this hypothesis matches the targeted automata. Submitting an equivalence query consists in asking the Oracle if the inferred grammar $\mathcal{G}$ is equivalent to the one of the targeted protocol. If not, it provides a counter-example that is used to correct the hypothesis. A counter-example takes the form of an input string $v \in (\Sigma_I')^*$ where $\lambda_{\texttt{SUL}}(v) \neq \lambda_{\texttt{HYP}}(v)$, with $\lambda_{\texttt{HYP}}(v)$ the result returned by our hypothesis state machine when it receives v. This process is iterated until no counter-example can be found.

Membership Queries

Membership queries are used to update the observation table while the table is not closed or not consistent.

L^* observation table is made of three parts (S, E, T) with $S \subset (\Sigma_I^{'})^*$ a nonempty finite set of *prefix closed* strings, $E \subset (\Sigma_I^{'})^*$ is a nonempty finite set of *suffix closed* strings and a finite function T mapping strings of $((S \cup S \cdot \Sigma^{'}) \cdot E)$ to strings from the output alphabet $\Sigma_O^{'}$. As reminded by M. N. Irfan in its thesis [70], a set is said *prefix closed iff* all the prefixes of every element of the set are also elements of the set. Conversely, a set is *suffix closed iff* all the the suffixed of every element of the set are also elements of the set. Furthermore if s is an element of $(S \cup S \cdot \Sigma_I^{'}\ )$, then row (s) denotes the finite function f from E to $(\Sigma_O^{'})^*$ defined by $f(e) = T(s \cdot e)$.

The observation table is consistent and closed if the two following properties are verified:

— The observation table is closed if all states reachable in one step from the inferred steps so far behaves like known states ($\forall t \in (S \cdot \Sigma'_I), \exists s \in S, row(t) = row(s)$).

— The observation table is consistent if all strings which lead to states thought to be equivalent so far have the same one-step behavior ($\forall s_1, s_2 \in S, row(s_1) = row(s_2) \Rightarrow \forall a \in \Sigma'_I, row(s_1 \cdot a) = row(s_2 \cdot a)$)

Searching for Counterexamples

In practical terms, the targeted implementation of a protocol is some kind of black box and equivalence queries must be approximated [130]. To achieve this, a solution consists in comparing results brought by representative membership queries submitted to both the implementation and the inferred model. Among existing testing methods used to pick these membership queries, the W-method [39] and its extension, the Wp-method [53] are often used. However, both of them have an exponential complexity in the size of the inferred automata. This complexity denotes the number of membership queries needed to estimate the equivalence of two automaton.

Results by Chian Y. Cho *et al.* [35] demonstrate the possibility to actively infer the communication protocol of a botnet modeled as a Mealy machine. To achieve this, the authors use the extension of the L^* algorithm, proposed by Niese [98], to infer Mealy machines. T. Bohlin *et al.* [20], T. Berg *et al.* [17] and F. Aarts *et al.* [2] reduced the inference complexity by representing an inferred grammar using a Symbolic Finite Automaton. In their model, messages having the same format and a similar impact over the grammar of the protocol are identified and abstracted with a single symbol. Thus, it requires less symbols than messages to represent the vocabulary of a protocol. This way, both the size of the vocabulary and of the inferred automaton can be reduced while preserving the grammar completeness.

Encouraged by these results, our work also relies on an active inference approach to learn the grammar of a protocol. This way, we can infer transitions that did not occurred in provided collected traces. However, previous works in this field have highlighted important issues that need to be addressed before inferring grammar of large protocols (*i.e.* protocols with numerous states and transitions). As we detail in the following, most of these issues are related to the submission of queries to an implementation.

As stated below, the key algorithm in the field of active inference is the L^* algorithm. When applied on large communication protocol inference, the execution of such algorithm can take an excessive amount of time. Indeed, D. Angluin [7] has shown that L^* can be used to infer a minimal DFA with a polynomial complexity in terms of membership queries $\mathcal{O}(|\Sigma|mn^2)$, m being the length of the longest counterexamples and n the number of states in minimal conjecture. For instance, T. Berg *et al.*, in their evaluation of the L^* algorithm [16], failed to learn some large protocols, such as ATM protocol, due to the excessive number of queries required to find counterexamples.

Reducing the number of queries and their length is the best strategy to limit the inference time. Indeed, each query takes the form of an exchange of several messages between the implementation and the inference algorithm. Moreover, it is mandatory to reset the implementation between each query. This operation can take a long time, especially if it implies the reboot of a computer system or resetting virtual machines. Both the number of queries and their length depend on the size of the targeted automata. Thus, inferring a large automata implies an important inference time.

Besides, we must also try to limit the number of messages sent to the implementation that does not imply any response. Indeed, we must use a timeout to detect such situation. If no message is received after a certain amount of time, *i.e.* the timeout period, we consider that no answer is triggered by our query. The repetition of this situation highly impact the overall inference time. Such situations are mostly due to the submission of invalid sequences of messages. It is thus important to limit the amount of queries that imply invalid sequences of messages.

Reversing proprietary protocols as those used by malware can be even more complex if the implementation detects the inference process and try to defend itself against it. To do that, it can rely on the frequency of received messages or more especially on the frequency of invalid messages. For example, the Ventrilo[3] proprietary protocol protects itself against reverse engineering attempts by counting invalid queries and banning clients that reaches a given threshold. An implementation can also introduce an additional delay before responding once a client has been detected as a threat. Another effective approach to fight against inference attempts consists in responding false information or to attack the client. A well known example of such protection is implemented in the Storm botnet [131] known to retaliate, through DDOS attacks, against any un-stealthy researcher.

By design, the L^* algorithm generates a large amount of invalid queries. This is a major limitation when applied on implementation that use such protection. This emphasizes the need to limit the amount of invalid sequences during active inference.

3. Ventrilo official web page: `http://www.ventrilo.com`.

Part I

Automated Inference of the Protocol Vocabulary

Chapter 4

Introduction

In this Part, we describe our work in the field of trace-based inference of protocol vocabulary. As described in Section 3.1, several studies were carried out to infer protocol's specifications from traces. Some of them rely on statistical measures to identify keywords that can be use to regroup messages [139, 83]. Others leverage sequence alignment algorithms to retrieve the field composition of messages and identify equivalent messages through the similarity of their syntax [14, 43]. Finally, some work also tend to consider message orders to improve the identification of equivalent ones [88]. As shown in our comparative study, these approaches do not provide accurate results on complex protocols and are often not applicable in an operational context to provide parsers or traffic generators, some key indicators of the quality of obtained specifications. In addition, too few previous works have resulted in the publication of tools that would allow the scientific community to experimentally validate and compare the different approaches.

In this thesis, we propose an approach to infer the specifications out of complex protocols by means of a semi-automated methodology. We believe that protocol reverse engineering is complex and in practical, its automated application on heterogeneous protocols may sometimes require the intervention of an expert. Thus, our objective is to automatize the bigger part of the inference process without refraining expert adaptations. To achieve this, we conceived a fine-grained vocabulary model and a methodology that automatically infers most parts of it by means of original techniques. Obtained results can latter be tuned by the expert.

The intuition behind our automated approach is that message classification and format inference are more effective if they also rely on the semantic definition of messages rather than only on their syntax, *i.e.* the sequence of static and dynamic fields. Unlike previous work, we use contextual information and its semantic definition as a key parameter in both the processes of message clustering and field partitioning. We also detect complex linear and nonlinear relationships between fields value, size and offset using correlation-based filtering. Besides, we propose multiple steps of clustering, each step leveraging a specific algorithm thus reducing the required computation time. We also show the viability of our approach through a comparative study including our reimplementation of three other state-of-the-art approaches (ASAP, Discoverer and ScriptGen). Finally, we have implemented our approach in Netzob [1] an open-source protocol RE framework we

1. Netzob: `http://www.netzob.org`

made available.

In the following, Chapter 5 describes our model of a protocol vocabulary. We then detail our automated inference methodology in Chapter 6. We conclude this part with our comparative study in Chapter 7 that evaluates our work and compares it to state-of-the-art approaches.

Chapter 5

Our Vocabulary Model

We describe in this Chapter the model we propose to represent the vocabulary of a protocol. As explained in Chapter 4 we created this model so its can be use to describe most communication protocols while being adapted to its automated inference. It covers multiple aspects of the vocabulary protocols such as the definition of its messages including their inner composition and the relationships that participate in their definitions. Our objective is to infer it in an automatic way. However, we also built our model so the expert can easily intervene and optimize the results of its inference. To achieve this, our fine-grained models exposes multiple features that can be leveraged by the expert to improve the results of the inference process. In the following, Section 5.1 exposes our definition of a symbol, its fields and of the values it accepts. We then explain in Section 5.2 the solution we propose to parse and generate valid messages according to this model. Finally, we detail in Section 5.3 our approach to represent the memorization strategy that is used by each field.

5.1 Symbols, Fields and Token-Tree

Similarly to the model proposed by F. Aarts [2], our model of a protocol vocabulary relies on symbols to represent messages that share the same format and have the same role from a protocol perspective. Nevertheless, we deviate from its work and propose to consider tokens as part of the definition of a symbol. As a reminder, we described in Section 2.2.1 that a token represents a succession of bytes that participate in the same meaning. Our objective is to ease our inference algorithm and to follow the definition of a protocol we exposed in Section 2.2.

In our work, a symbol is composed of a succession of fields. However, we believe that fields are difficult to infer as they can be made of different tokens that only share a semantic equivalency. For example, HTTP requests and responses include a *version field* that describes the major and the minor version number of the HTTP protocol used to create the message. In its specifications [49, 96], values accepted by this field are described under an ABNF notation. Listing 5.1 reminds it.

```
HTTP-Version   = "HTTP" "/" 1*DIGIT "." 1*DIGIT
```

Listing 5.1– ABNF definition of the HTTP version number field as described in RFC 2616 [49])

This field accepts values such as “HTTP/2.4” or “HTTP/12.3”. Indeed, as shown by its specifications, this field is made of five different tokens: “HTTP”, “/”, one or more digits, “.” and one or more digits. We therefore propose to refine the definition of a field by introducing the notion of *token-tree* that represents the set of tokens it accepts. Thus, we model a symbol as a succession of fields and a field as a composition of tokens described by a token-tree.

In the remainder, Section 5.1.1 describes our definition of a symbol and Section 5.1.2 our one of a field. We finally detail our definition of a token-tree in Section 5.1.3. We illustrate these definitions through practical examples of protocol vocabularies we describe with the language offered by Netzob.

5.1.1 Definition of a Symbol

As stated in section 2.2, the vocabulary of a protocol defines the set of messages its accepts. In the remainder, Σ denotes the vocabulary of a protocol accepted by one of its implementation. This vocabulary is divided into two subsets, $\Sigma = \Sigma_I \cup \Sigma_O$ where Σ_I denotes all the messages the implementation can receive and Σ_O all the messages it can send. Input messages refer to the former and output messages for the latter. As shown in previous work dealing with vocabulary inference [20, 17, 2], exchanged messages in a protocol often include parameters. Given that certain parameters can take their values in a theoretically infinite definition domain or can depend on the value of others, there can be an infinite number of messages. To represent messages in a more compact model, we use symbols to abstract similar messages from a protocol perspective that only differ from the values of their parameters. We denote $\Sigma^{'}$ the symbolic vocabulary of a protocol and $\Sigma^{'} = \Sigma^{'}_I \cup \Sigma^{'}_O$ with $\Sigma^{'}_I$ and $\Sigma^{'}_O$ the set of input and output symbols of the protocol.

Definition A *Symbol* is the common abstraction of multiple messages, sharing the same format and having the same role from a protocol perspective. By format, we hereby refer to a sequence of fields. We denote Σ' the set of all possible symbols accepted by a protocol, *a.k.a* its symbolic vocabulary and $s^x \in \Sigma'$, the symbol with role x. For example, the set of `DHCP DISCOVER` messages can all be abstracted by the same symbol $s^{\texttt{DISC}}$. An `ICMP ECHO REQUEST` and `SMTP EHLO` commands are other kinds of symbols respectively denoted $s^{\texttt{ECHO-REQ}}$ and $s^{\texttt{EHLO}}$.

5.1.2 Definition of a Field

To describe the composition of a symbol in fields, we use the notation introduced by F. Aarts [2]. A symbol s^x follows a format that specifies a tuple of fields denoted $s^x = (f^x_0, f^x_1, ..., f^x_n)$ with f^x_i defined in $\mathcal{F}$ the set of all possible fields. We represent the values that are accepted by a field under a grammatical form we call a *token-tree*. We provide a definition of a token-tree in Section 5.1.3.

Definition A *field* is the common abstraction of a set of tokens that share a common meaning from a protocol perspective. This meaning is established by the protocol creator. A symbol is made of a succession of fields and each field can either accept a unique or multiple values. For example, Figure 5.1 illustrates the set of fields that could be use to model the composition of an HTTP symbol.

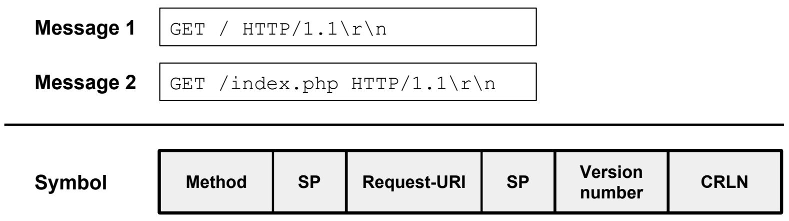


Figure 5.1 – Example of fields that can be use to model an HTTP symbol

A field can either accept a unique constant value or a set of different values. In our example, the field $f^{\texttt{HTTP}}_{\texttt{Request URI}}$ accepts at least two different values "/" or "/index.php" while $f^{\texttt{HTTP}}_{\texttt{SP}}$ only accepts "␣". Thus, our vocabulary model accepts three types of fields, 1) static fixed-size fields, 2) dynamic fixed-sized fields and 3) dynamic variable-sized fields.

A **static fixed-size field** or static field accepts a single constant value which by definition is fixed-size. For example, all the requests and replies in the SMB protocol starts with the same sequence of bytes, *i.e.* `0xFF534D42`.

A **dynamic fixed-size field** denotes a field that accepts different values that have the same size. Many binary protocols include such fields. For example, the "version" field present in every IP message is a fixed-size field that accepts different values (*e.g.* `4` to represent an IP message or `8` to represent a PIP [51] message.)

A **dynamic variable-size field** is the most complex type of field to parse and to infer. It accepts multiple values that can be of different sizes. We can find a lot of these fields in ASCII protocols. For example, the HTTP "version number" field accepts different values of different sizes such as `HTTP/1.1` or `HTTP/1.11`.

5.1.3 Definition of a Token-Tree

We previously explained that a field can accept multiple values and that a value can be made of different tokens. To model these tokens, we attach to each field a token-tree. This token-tree models the definition domain of its field, *i.e.* the set of values it accepts. A token-tree is an ordered, rooted tree that represents the syntactic structure of tokens that are accepted by its field. It follows a n-ary *right-branching* structure [15] that grows downward and proceeds left to right. Similarly to a Constituency-based parse tree [], a token-tree distinguishes between non-terminal and terminal nodes. A non-terminal node is a node that has one or more children nodes, either non-terminal or terminal ones. A terminal node is a node that has no children and can be seen as a leaf of the tree. Besides, nodes in a token-tree are labeled. Terminal nodes are either labeled with **static** or **dynamic** tokens while non-terminal nodes are either labeled as **aggregates**, **alternates** or **repeats** nodes. We detail these labels in the following and illustrate them by means of examples including sample usages of **Netzob**.

Definitions of Terminal Nodes accepted by a Token-Tree

A **static token** (*e.g.* a magic number in a protocol header) labels non-terminal nodes. It represents a single constant value. To model a field which accepts a single constant value, we attach to it a token-tree that contains a single terminal node labeled with a static token. For example, listing 5.2 illustrates the token-tree of a field $f0$ that is made of a unique terminal node labeled with a static token which value is "helloworld".

```
>>> # defines f0, a field which only accepts "helloworld"
>>> f0 = Field("helloword")
```

Listing 5.2– Example of a static token

A **dynamic token** (*e.g.* the username field in the IRC protocol) represents a set of values that share the same type and the same size range. Thus, a dynamic token is described with a type and a size. We support various token types such as ASCII, decimal, IPv4, raw byte or bit array. We use a range to describe the minimum and the maximum size in bits of the values a dynamic token accepts. For example, Listing 5.3 represents two field $f0$ and $f1$. The former accepts any sequence of four bytes while $f1$ accepts any string of ten to twenty ASCII chars.

```
>>> # f0 is a field that accepts any sequence of 4 bytes
>>> f0 = Field(Raw(nbBytes=4))

>>> # f1 is a field that accepts any ASCII string of 10 to 20 chars
>>> f1 = Field(ASCII(nbChars=(10,20)))
```

Listing 5.3– Example of a dynamic token

Besides its size and its type, an additional constraint can be added to the definition of a dynamic token. This constraint can be use to model a relationship between its value and the value or the size of one or more other fields. Our model accepts three types of relationships: 1) *intra-symbol relationship*, 2) *inter-symbol relationship* and 3) *environmental relationship*.

An *intra-symbol relationship* describes a relationship between a token and one or more fields that participate in the same symbol. For example, such relationship can be use to model a CRC32 field. To represent this constraint, we use a function taking as parameter some fields of the same symbol. Based on our observation of common protocols, we identified two recurrent functions: 1) $size : \mathcal{F}^* \rightarrow \mathbb{N}$ a function that returns the size in bits of one or more consecutive fields and 2) $value : \mathcal{F}^* \rightarrow \mathbb{B}$ a function that returns the value of one or more fields. $\mathbb{B}$ represents all the possible sequence of bits $b \in \Sigma_{0,1}^*$ and $\Sigma_{0,1} = \{0, 1\}$. These functions can be combined with common mathematical operations to define, for example, that a field contains the CRC32 of another field. Listing 5.4 shows the specification of an intra-symbol relationship in Netzob.

```
>>> # f1 is a dynamic variable-size field of 0 to 30 chars.
>>> f1 = Field(ASCII(nbChars=(0,30)))
>>> # f0 is a dynamic fixed-size field which value is the size of f1
>>> f0 = Field(Size([f1], nbBytes=2))
>>> # create a symbol composed of fields f0 and f1
>>> s = Symbol(fields=[f0, f1])
```

Listing 5.4– Example of an intra-symbol relationship

An *inter-symbol relationship* describes a relationship between a token and one or more fields that belong to a previous symbol transmitted during the same session. For example, such relationship exists in the TCP protocol to define the value of an acknowledgment number. We use the same functions than for intra-symbol relationship but specify as parameters fields of other symbols. Listing 5.5 illustrates the specification of an inter-symbol dependency in Netzob.

```
>>> # f1 is a dynamic variable-size field of 0 to 30 chars.
>>> f1 = Field(ASCII(nbChars=(0,30)))
>>> s1 = Symbol(fields=[f1])

>>> # f0 is a dynamic fixed-size field which value is the size of f1
>>> f0 = Field(Size(f1, dataType=Raw(nbBytes=2)))
>>> s0 = Symbol(fields=[f0])
```

Listing 5.5– Example of an inter-symbol relationship

Finally, the values of a dynamic token can also be constrained by an *environmental relationship*. Such relationship specifies that the value of a token depends on an environmental property such as the current IP source, the date or the hostname. Similarly to inter and intra symbol relationships, an environmental relationship is represented by a function $Env : \mathbb{E} \rightarrow \mathbb{B}$ that takes as parameter the name of an environment property, $e \in \mathbb{E}$. For example, Listing 5.6 illustrates a field that takes as value the current hostname of the system.

```
>>> # f0 is a dynamic variable-size field that contains the message
    author hostname
>>> f0 = Field(Env("hostname"))
```

Listing 5.6– Example of an environmental relationship

Definitions of Non-Terminal Nodes accepted by a Token-Tree

Multiple static and dynamic tokens can be combined to form a complex and precise specification of the values that are accepted by a field. A combination is modeled by non-terminal nodes in the token-tree of a field. We propose the use of three different combinations: 1) aggregate, 2) alternate and 3) repeat. We detail them in the following.

An **aggregate** node concatenates the values that are accepted by its children nodes. It can be use to specify a succession of tokens. For example, Listing 5.7 represents a field which accepts values that are made of an ASCII of 3 to 20 random characters followed by a ".txt" extension.

```
>>> # Specifies a field made of two aggregated tokens
>>> t1 = ASCII(nbChars=(3,20))
>>> t2 = ASCII(".txt")
>>> f = Field(Agg([t1, t2]))
```

Listing 5.7– Example of a field which definition domain is an aggregation of two tokens

Tokens can also be combined under an alternative form. This combination is represented by an **alternate** node. It can be seen as an OR operator between two or more children nodes. For example, listing 5.8 denotes a field accepts either "*filename1.txt*" or "*filename2.txt*".

```
>>> # Specifies a field made of two alternate tokens
>>> t1 = ASCII("filename1.txt")
>>> t2 = ASCII("filename2.txt")
>>> f = Field(Alt([t1, t2]))
```

Listing 5.8– Example of a field which definition domain is an alternate of two tokens

Lastly, a field can also be defined under a repetition form of one or multiple tokens with a **repeat** non-terminal node. It denotes an n-time repetition of a terminal or a non-terminal node. For instance, we can use this operation to specify a field which token-tree accepts a repetition of n IPv4 addresses where n is the value of another field. Listing 5.10 shows such symbol made of two fields, the former contains the number of IPv4 addresses that are declared in the second field. The repeat operator is used to represent a dynamic number of IPv4 tokens in a single field.

```
>>> f1 = Field(Decimal(interval=(1,5)))
>>> f2 = Field(Repeat(IPv4(), nbRepeat=value(f1)))

>>> # Creation of a symbol composed of these two fields
>>> s = Symbol(fields=[f1, f2])
```

Listing 5.9– Example of a field which definition domain is a repetition of IPv4 addresses

In this Section, we presented how we specify a symbol, its fields and the grammatical representation of the values they accept. In the following, we detail the process we use to verify that a message is valid according to the definition of a symbol. We refer to this process as the *abstraction*. We also explain how we *specialize* a symbol to generate valid messages according to its definition.

5.2 Abstraction and Specialization

As presented above, the use of a symbolic model is required to represent the vocabulary of a protocol in a compact way. However, the objective of this thesis is also to infer the grammar of the communication protocols which implies, in our case, to exchange messages with an implementation of a protocol. We therefore need to abstract received messages into symbols that can be used by our model. Conversely, we also need to specialize symbols produced by our model into valid messages. To achieve this, we use an *abstraction* block and a *specialization* block. As illustrated on Figure 5.2, these blocks play the role of an interface between our symbolic model and a communication channel.

To compute or verify the relationships that participate in the definition of fields, we include in our model a memory. This memory stores the value of previously captured or emitted fields. It takes the form of *state variables*, one for each field of the protocol. Each state variable, $v \in \mathcal{V}$ stores the current value of its field. For example, if the definition domain of a field f_0^x denotes an equality relationship with another field f_1^x, any modification to the value of f_0^x state variable, denoted $v(f_0^x)$, is automatically passed on the value of f_1^x state variable. This relationship between state variables is asymmetric thus, in our example, a modification of f_1^x does not trigger the modification of f_0^x. Initially, a memory is created with an undefined state variable for each dynamic field. However, if a field only contains static tokens optionally combined under various aggregated node, its constant value is initially stored in the state variable. In the following, we give our definition of the abstraction and specialization blocks and describe their usages of the memory.

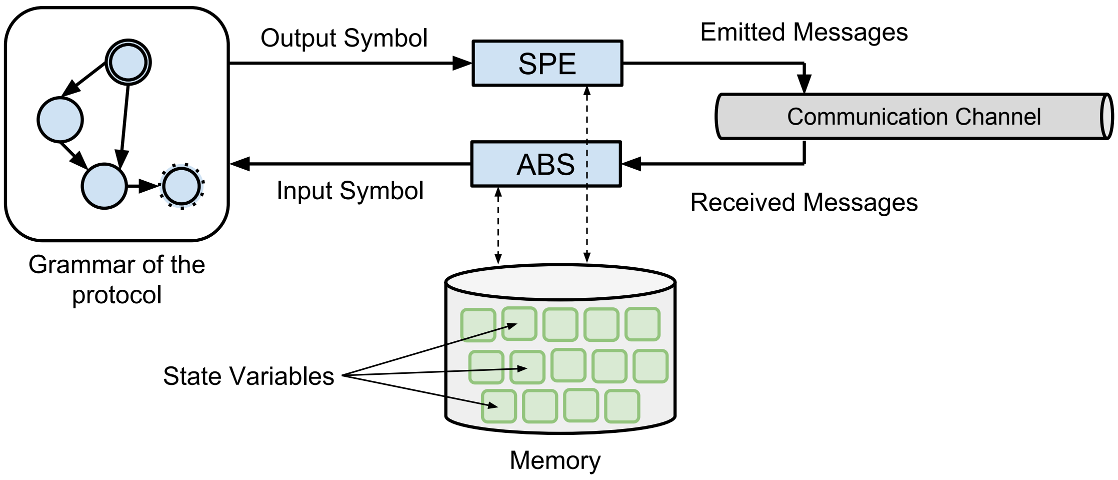


Figure 5.2 – Abstraction (ABS) and Specialization (SPE) blocks.

We use the term of *abstraction* to denote the transformation of a message into a symbol. Given a message, this operation checks if the value of its fields complies with the definition domain of the symbol fields. If requested, it also memorizes the received field values to ensure the computation or the verification of relationships. More formerly, this operation is represented by the function ABS : $\Sigma_I \times \mathcal{V} \rightarrow \Sigma_I^{'} \times \mathcal{V}$, that given a received message $m \in \Sigma_I$ and current state variables values $\bar{v}$, returns the associated symbol $s^x \in \Sigma_I^{'}$ and a new state vector.

Conversely, we use the term of *specialization* denoted by the function SPE : $\Sigma^{'} \times \mathcal{V} \rightarrow \Sigma \times \mathcal{V}$ to define the transformation of a symbol into a message. This function returns a message $m \in \Sigma$ given the current state vector $\bar{v}$ and a symbol $s \in \Sigma^{'}$. In practical, this operation builds a message by successively specializing each field of the symbol. By definition, the specialization of a field consists in returning the value stored in its state variable. If the state variable is undefined, we first generate a new value based on its definition domain and saves the value in its state variable. We then use this new value in the message. Thus, similarly to the abstraction function, the specialization of a symbol also returns a new vector of state variable values. The vector represents the memory after processing the message.

The use of a memory requires to specify how and when this memory is accessed. Up to here, previous work that support field relationships [43, 88] always assumed that the abstraction and the specialization of a field follows a default memorization strategy, *i.e.* each abstracted field value is memorized while each specialized field value is generated using the value of its corresponding state variable. This default strategy does not apply on all the fields. For example, in the IRC protocol the `MSG` symbol that can be used to send a private message to a user or a to channel, requires different memory usage. This symbol is made of, at least, two fields. The first field contains the destination of the message while the second field contains the content of the message. When specializing the `MSG` symbol, the destination field may be filled with a previously observed user or channel name to be valid, while the content field may be filled with ASCII values that must be generated every-time the `MSG` symbol is sent. To achieve this, the specialization of the first field requires to use the memorized value stored in its state variable while the specialization of the second field requires to generate new content according to its definition domain. Such different strategies can also be found in the abstraction process. Thus, to model how and when each field is memorized, we propose the use of a State Variable Assignment Strategy (SVAS).

5.3 State Variable Assignment Strategy (SVAS)

As stated below, our model includes a memory to ensure the computation and the verification of relationships between fields. This memory relies on state variables to store the value of each field and is managed by a strategy, we call the State Variable Assignment Strategy (SVAS). A SVAS is attached to each field and is used both when abstracting and specializing the field. This strategy describes the set of memory operations that must be performed every time a field is abstracted or specialized. These operations can be separated into two groups, those used during the abstraction and those used during the specialization. From our observation of common protocols, we identified two abstraction operations and three specialization operations. In the following, we first describe the two operations that can participate in the SVAS of a field when abstracting it: **ValueCMP** and **Learn**. It must be noted, that these operations are only executed if the received values complies with the field's definition domain.

The **ValueCMP** operation compares the value of a received message field against the memorized value stored in the associated state variable. This operation checks if both are equals and if not, stops the message abstraction process. For example, this operation can be use to ensure an equality relationship.

The **Learn** operation saves the value of a received message field into its corresponding state variable. This way, the saved value can latter be used to abstract or to specialize other fields. Learning an already defined state variable override its current value with the new one. Typically, this operation can be used to abstract the sequence ID field in the TCP protocol, which needs to be reused to generate or abstract another TCP message.

In addition to these abstraction operations, the SVAS of a field also describes the impact of the specialization over the memory. From our observation of protocols, we identified three different operations that can participate in the SVAS of a field when we specialize it: **Use**, **Regenerate** and **Memorize**.

The **Use** operations reads the value stored in the field's state variable and uses it as the field value in the message. For example, this operation can be use with an equality relationship that synchronizes two state variable's value.

The **Regenerate** operation generates a new value that respects the definition domain of the field. For example, sending a TCP message requires to generate a new valid TCP sequence ID field. This operation has no effect over the state variable of the field but can be coupled with the **Memorize** operation.

The **Regenerate** operation generates a new value that respects the definition domain of the field. For example, sending a TCP message requires to generate a new valid TCP sequence ID field. This operation has no effect over the state variable of the field but can be coupled with the **Memorize** operation.

The **Memorize** operation is similar to the **Learn** operation but is used during the specialization process to save in memory the emitted value of a field. This operation can be attached to the **Regenerate** operation to store (or override) in memory a newly generated value. In the case of a TCP message, we use this operation to memorize the generated sequence ID in order to ensure the validity of the next received or sent TCP message.

To illustrate the definition of a SVAS, we take as example the ICMP protocol. In its grammar, this protocol includes a transition triggered by the reception of an echo symbol (`ECHO`) that responds with an echo reply symbol (`REPLY`). Both the two symbols have a data field. As stated by the RFC 792, "The data received in the echo message must be returned in the echo reply message" [1]. We represent this with an inter-symbol relationship that specifies that the value of the $f_{data}^{\texttt{REPLY}}$ equals to the value of the $f_{data}^{\texttt{ECHO}}$ relationship ensures that every-time the state variable of the $f_{data}^{\texttt{ECHO}}$ is modified, the same modification is applied on the state variable of $f_{data}^{\texttt{REPLY}}$. Figure 5.3 illustrates the different operations we use to model this relationship. In this figure, we use the notation `$v_NAME` to represent the state variable assigned to the field `NAME` and `$NAME` the current value of the field `NAME` in a message.

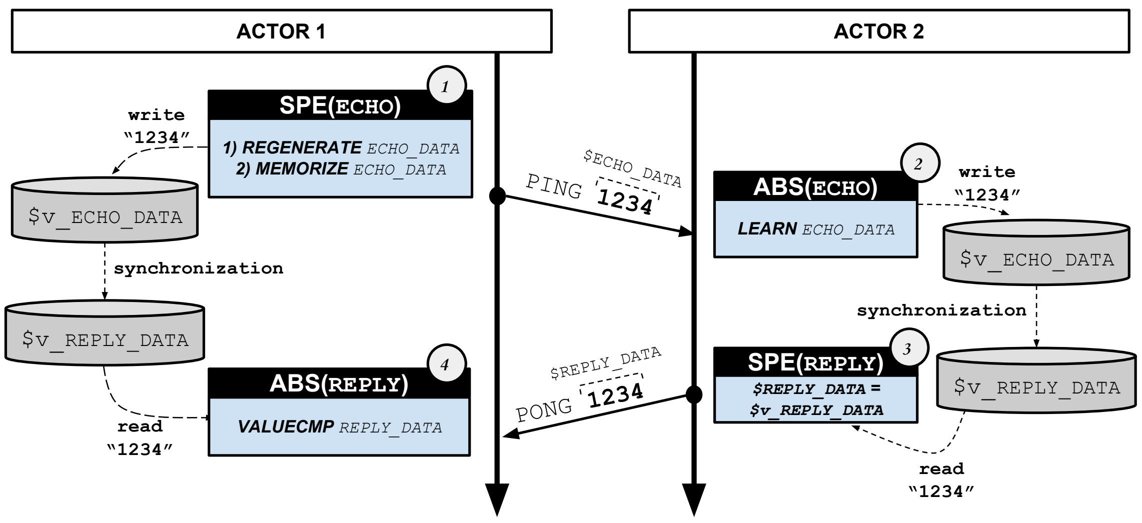
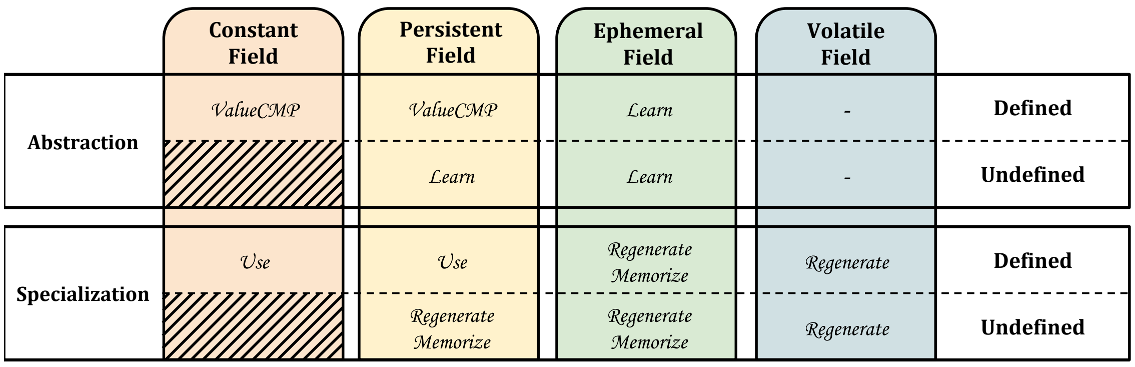


Figure 5.3 – Memory operations for both the abstraction (ABS) and the specialization (SPE) of ICMP echo-request exchanges.

1. RFC 792: Internet Control Message Protocol. p.14

First, every time we emit an `ECHO` symbol, its data payload contained in field $f_{data}^{\texttt{ECHO}}$ must be generated (operation **Regenerate**) and memorized (operation **Memorize**). This way, Actor 1 is able to verify that the received value of the data payload in the response message is the same. When abstracting the received `ECHO` message, the Actor 2 saves (operation **Learns**) the received data field contained in $f_{data}^{\texttt{ECHO}}$. This value is then reused to specialize the `REPLY` symbol (**Use** operation). Finally, when Actor 1 receives and abstracts the `REPLY` symbol, he compares (operation **ValueCMP**) the value with the memorized one. If they are equals, the received message is a valid reply message and the ICMP exchange complies with the RFC.

To ease the specification of a field, we simplified our model by identifying common behaviors in communication protocols and propose four different type of fields, each denoting a typical SVAS: **constant fields**, **persistent fields**, **ephemeral fields** and **volatile fields**. Figure 5.4 illustrates those different SVAS. It sums-up the different memory operations that are used depending on the type of fields and on the definition status of their corresponding state variable. "–" represents that no operation related to the memory is performed. However, we remind that in any cases, abstracting a received value with a field requires first that its value complies with the field definition domain.



	Constant Field	Persistent Field	Ephemeral Field	Volatile Field	
Abstraction	*ValueCMP*	*ValueCMP*	*Learn*	-	**Defined**
	▨	*Learn*	*Learn*	-	**Undefined**
Specialization	*Use*	*Use*	*Regenerate Memorize*	*Regenerate*	**Defined**
	▨	*Regenerate Memorize*	*Regenerate Memorize*	*Regenerate*	**Undefined**

Figure 5.4 – Our SVAS template that models the memory operations performed while abstracting and specializing a field.

A **constant field** is a very common type of field as it denotes a static content defined once and for all in the protocol. By nature, this state variable is always defined. When abstracting such field, its value is compared against the value of the corresponding variable (**ValueCMP**). On the other hand, the specialization of a constant field does not imply any additional operations than using the memorized value as field value (**Use**). A typical example of a constant field is a magic field or a delimiter field.

A **persistent field** carries a value, such as a session identifier, generated and memorized during its first specialization and reused as such in the remainder of the session. To model this behavior, we rely on the definition status of its state variable. During its first specialization, the corresponding state variable is undefined, and so a new value is generated (**Regenerate**) and memorized (**Memorize**). If the same symbol is specialized latter in the session, the corresponding state variable is now defined and we use it as a field value (**Use**). Conversely, the first time such persistent field is abstracted, its state variable is not defined and the received value is saved (**Learn**). Latter in the session, if this field is abstracted again, the corresponding variable is now defined and we compare (**ValueCMP**) the received field value against the memorized one.

The value of an **ephemeral field** is regenerated (**Regenerated**) every time it is specialized. The generated value is memorized in its corresponding state variable to abstract or specialize other fields. During abstraction, the value of this field is always learned (**Learn**) for the same reason. The IRC NICK command includes such ephemeral field that denotes the new nick name of the user. This nick name can afterward be used in other fields but whenever a NICK command is emitted, its value is regenerated.

Finally, a **volatile field** denotes a value which changes (**Regenerated**) whenever it is specialized and that is never memorized. It can be seen as an optimization of an **ephemeral field** to reduce the memory usages. Thus, the abstraction process of such field only verifies that the received value complies with the field definition domain without memorizing it. For example, a size field or a CRC field are volatile fields.

In our previous example of the ICMP protocol, we use en equality relationships between state variables of both f^{ECHO}_{data} and $f^{\mathrm{REPLY}}_{data}$ to synchronize their values. To model their memory strategy, we use an **ephemeral field** for f^{ECHO}_{data} field and a **persistent field** for $f^{\mathrm{REPLY}}_{data}$.

```
>>> #
>>> # ICMP ECHO REQUEST SYMBOL
>>> #
>>> pingHeaderField = Field(name="Header")
>>> pingTypeField = Field(name="Type", domain=Raw('\x08'))
>>> pingCodeField = Field(name="Code", domain=Raw('\x00'))
>>> pingCksumField = Field(name="Checksum")
>>> pingHeaderField.children = [pingTypeField, pingCodeField,
    pingCksumField]

>>> # set the checksum field
>>> pingCksumField.domain = crc32(pingHeaderField)

>>> # create the ping data field
>>> pingDataField = Field(name="Data", domain=Raw(nbBytes=(8,80), type=
    EPHEMERAL)

>>> # create the ping request symbol
>>> pingSymbol = Symbol(fields=[pingHeaderField, pingDataField])

>>> #
>>> # ICMP ECHO REPLY SYMBOL
>>> #
>>> pongHeaderField = Field(name="Header")
>>> pongTypeField = Field(name="Type", domain=Raw('\x00'))
>>> pongCodeField = Field(name="Code", domain=Raw('\x00'))
>>> pongCksumField = Field(name="Checksum")
>>> pongHeaderField.children = [pongTypeField, pongCodeField,
    pongCksumField]
```

```
>>> # set the checksum field
>>> pongCksumField.domain = crc32(pongHeaderField)

>>> # create the pong data field
>>> pongDataField = Field(name="Data", domain=value(pingDataField),
    type=PERSISTENT)

>>> # create the pong request symbol
>>> pongSymbol = Symbol(fields=[pongHeaderField, pongDataField])
```

Listing 5.10– Usage of persistent and ephemeral fields to specify ICMP symbols

Chapter 6

Leveraging Semantic Information to Improve the Vocabulary Inference

6.1 Introduction

This chapter details our approach to infer the vocabulary of an unknown protocol. The intuition behind our work is that message classification and format inference are more effective if they also rely on the semantic definition of messages rather than only on their syntax, *i.e.* the sequence of static and dynamic fields. In our approach, we identify the semantic associated to a given part of a message. We then take this information into account both to identify fields and to cluster messages.

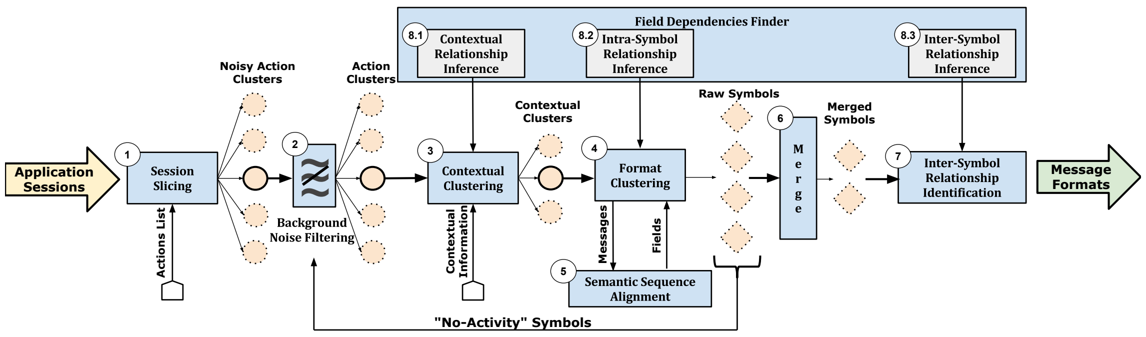


Figure 6.1 – System overview

As illustrated in figure 6.1, we take as inputs some traces collected upon the execution of a set of user actions over the protocol implementation and use them to infer the protocol specifications. We use the term of **application session** to mean these traces. The idea is to consider the semantic definition at every steps of message clustering and partitioning. Thus, our approach relies on several sub-steps participating in pre-clustering steps (*cf.* blocks 1, 2 and 3 in figure 6.1), the main clustering step (*cf.* blocks 4 and 5), the merging step (*cf.* block 6) and the inter-symbols relationships inference step (*cf.* block 7). The different pre-clustering steps aim at computing homogeneous clusters, which is mandatory to obtain good results during the sequence alignment. However, this approach can generate too many similar clusters. Thus, the goal of the merging step is to combine clusters that share the same format.

The rest of this Chapter is organized as follows. Section 6.2 describes our solution to collect semantic information while capturing sample traces of the targeted protocol. We then explain in Section 6.3 our solution to leverage this semantic to improve message clustering and to uncover fields definitions. Finally, Section 6.4 denotes our work to automatically infer relationships between fields.

6.2 Collecting Semantic Information

Our approach not only requires messages that belong to different application sessions but also semantic information associated to these sessions. By semantic information, we refer to 1) the definition of actions performed by the implementation during the capture, each denoting the execution of a specific feature of the protocol, *e.g.* "List Directory", "Read File", "Write File" in an FTP session and 2) the contextual data accessed by the implementation while executing these actions. We use actions in the Sessions Slicing step detailed in section 6.3.1 while we leverage contextual data in our Contextual Clustering step detailed in section 6.3.3. To collect this semantic information, we use three different solutions, depending on the control we have on the capture process.

The first solution applies when we have access to an implementation of the protocol that exposes either a graphical or a command line interface. In that case, we first identify the different input parameters offered by the interface. Based on this, we establish various scenarios each implying different actions to perform with arbitrary predefined parameters. We then capture the communications resulting from the automatic execution of these scenarios. To do this, we can rely on scripts or on graphical interface testers such as Sikuli [145]. This solution is the most effective to collect both the actions performed by the implementation and the contextual data.

We use the second solution when we have no control over the implementation of the protocol but we can monitor its execution. This situation arises when reversing protocols used by malware. This solution relies on the instrumentation of the OS on which the implementation is executed. In practice, we use sandboxes, such as Cuckoo [40], to capture network traffic in parallel with any useful contextual information related to the application, such as names of accessed files, network parameters or system calls. To retrieve the actions associated to the generated traffic, we identify the different actions based on captured system calls. Indeed, we believe that a signature of successive system and function calls can distinctly represents a specific program action on a system. For example, we have successfully instrumented the Android Dalvik Virtual Machine with the Substrate Framework [52] to intercept and collect, at runtime, any contextual information accessed by an application, such as Android version number, phone contacts, SMS providers. We also monitor specific API calls denoting the execution of actions, *e.g.* the creation of activities, the activation of devices such as GPS or bluetooth, reading and writing personal user information.

If we have no access to the implementation, *e.g.* when only traces are provided by third parties, we have to manually specify contextual information. In addition, environmental information is also automatically extracted from files meta-data, such as IP addresses, hostnames and TCP/UDP port number.

6.3 Semantic-based Message Clustering

In this section, we describe the whole process of message clustering and detail the different steps of this process. The goal is to compute a common abstraction model, *i.e.* a symbol, for similar messages.

6.3.1 Session Slicing (Step 1)

This first step in our clustering process consists in slicing sessions into *action frames* denoting different actions. This step is based on the two following heuristics. At first, we believe that initiators, *i.e.* actors sending the first message, and non-initiators of a communication can use two different subsets of the vocabulary. Secondly, we observe that most of the different messages are linked to specific actions. Hence, identifying these action frames allows us to find and pre-cluster messages implied in the same action but captured in different application sessions or at different times in the same session.

To perform this step, message timestamps are compared against the start time of each action. Messages with a timestamp superior or equal to the start time of action n and inferior to the start time of action $n+1$ are clustered together. Obtained clusters are then subdivided into to sub-clusters, one for *sent messages* and the other for *received messages*. By sent message, we refer to a message sent by the initiator of the communication channel while a received message denotes a message received by it. For example, Figure 6.2 illustrates tree collected sessions. Each trace is made of several received and sent messages. We obtained the first two sessions while executing three actions $\mathcal{A}_1$, $\mathcal{A}_2$ and $\mathcal{A}_3$. Conversely, no actions were performed while capturing the third session. Based on action timestamps, we slice the first two sessions and cluster their messages according to their action frames and if they were sent or received. However, messages of the third sessions are related to no action. Thus, we divide them into two clusters, one for the sent messages and the other for received ones. In our example, messages m_1, m_3, m_5 and $m_5^{'}$ are grouped into the $\mathcal{A}_1^{\texttt{Sent}}$ cluster while messages m_2, m_4 and $m_6^{'}$ belong to the $\mathcal{A}_1^{\texttt{Received}}$ cluster.

When no information is available regarding the underlying actions performed by the application while traces were captured, we rely on a statistical analysis to detect action frames. We approximate action frame boundaries using variations of the inter-arrival time. Originally developed by U. Gargi [54] to cluster collections of pictures following their timestamps, this approach defines the following heuristics: 1) a long interval with no information usually marks the end of an action; and 2) a sharp upward change in the frequency of information inter-arrival time usually marks the beginning of a new action.

This step produces different *action clusters*, one for each type of action and for each type of actor (initiator or not). Among these action clusters, two can represent messages that are related to no action. In the next Section, we describe how we leverage these clusters to filter background noise from the other clusters.

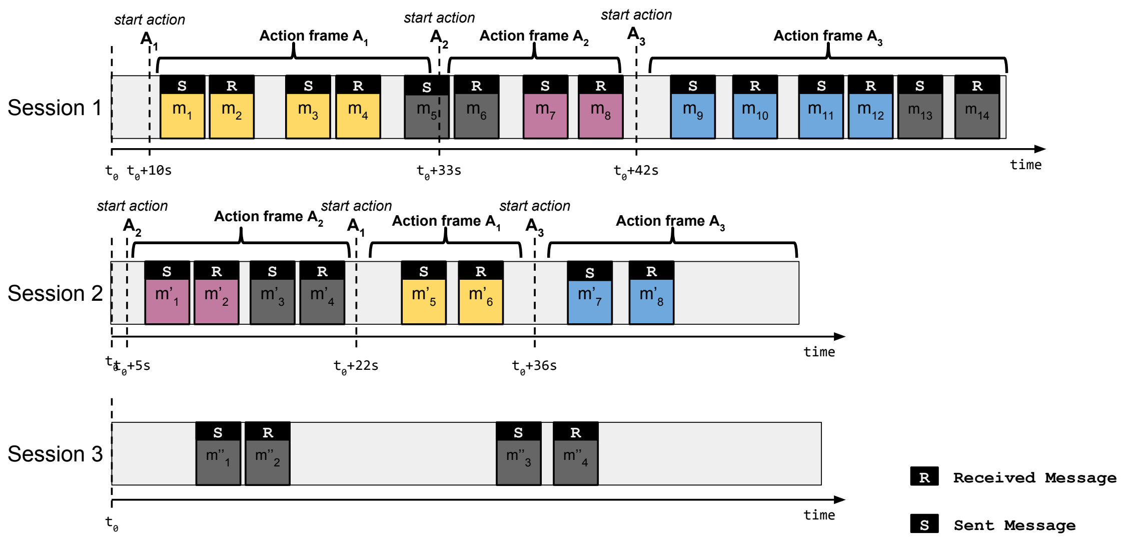


Figure 6.2 – Examples of session slicing.

6.3.2 Background Noise Filtering (Step 2)

Some messages are not correlated to a particular action and can occur at any time, for example the PING and PONG messages used in IRC protocol. These messages correspond to background noise. As they have their own format, which is generally different from the format of messages generated by user actions, we need to filter them from *action clusters*. To do so, we rely on sent and received messages during a period of no activity.

Filtering background noise is achieved in two steps. First we execute the contextual clustering (step 3) and format clustering (step 4) on messages belonging to the no activity clusters. As detailed in the following sections, these algorithms create a symbol for each type of messages belonging to background noise. In a second step, we filter each *action cluster* using these symbols. If a message can successfully be parsed with a noise symbol, we remove it from the *action cluster*.

For example, the execution of this filtering process refines the clusters resulting of the action frame slicing of sessions presented in Figure 6.2). Indeed, messages m_5, m_{13} and $m_3^{'}$ were wrongly clustered in sent action clusters. This filtering process identifies received and sent messages to regroup them in two dedicated no-action clusters labeled $\mathcal{B}^{\texttt{Sent}}$ and $\mathcal{B}^{\texttt{Received}}$ on Figure 6.3.

6.3.3 Contextual Clustering (Step 3)

Action clusters can still suffer from imprecision, as a single action frame can contain messages of different formats. For instance, a single user action such as the connection to an SMB share directory generates 16 different messages with smbclient [1].

The contextual clustering step refines *action clusters*. The main idea is to regroup messages that embed the same type of contextual information, such as host addresses, timestamps or usernames.

1. smbclient is a “client to access SMB/CIFS resources on servers” and is developed by the Samba Team.

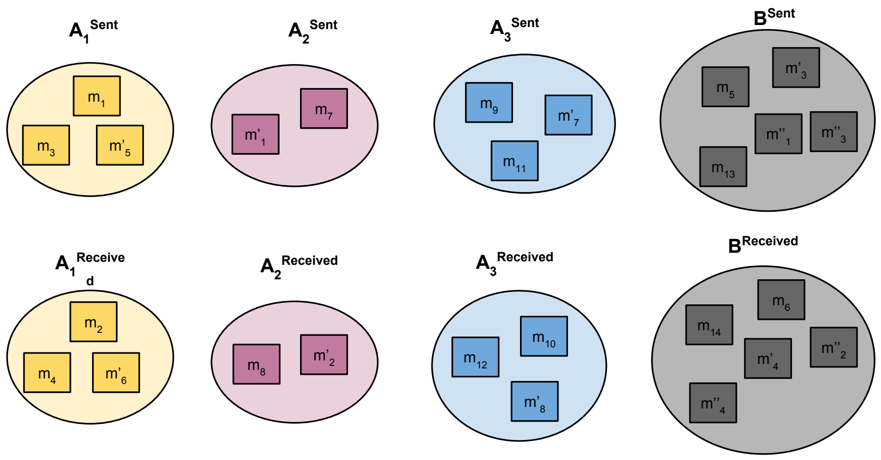


Figure 6.3 – Action Clusters resulting in the background noise filtering process.

To achieve this, we rely on contextual data collected during the capture or extracted from the capture files meta-data. We search occurrences of these data in every messages of a given action frame. We finally subdivide action clusters by grouping messages that share the same sequence of type of contextual data. For example, Figure 6.4 illustrates the contextual clustering process applied on the action cluster $\mathcal{A}_1^{\texttt{Sent}}$. It searches for contextual information in its three messages, and identifies the presence of the destination IP and of the username in them. The former information was extracted from the trace file (*e.g.* a *pcap* file) while the latter is provided by the expert. Messages that embed the same type of contextual information are regrouped into the same clusters. In our example, two clusters are created: $\mathcal{A}_{1,1}^{\texttt{Sent}}$ and $\mathcal{A}_{1,2}^{\texttt{Sent}}$.

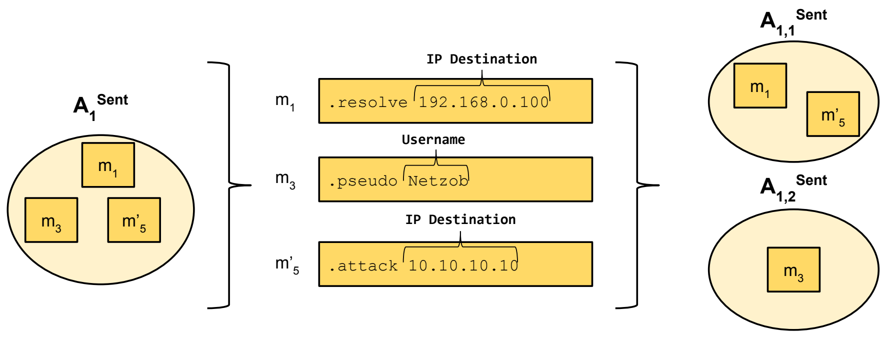


Figure 6.4 – Illustration of the contextual clustering process.

This contextual information is searched in messages using different encoding (*e.g.* little-endian, big-endian, ASCII, UTF-16) and common transformations (*e.g.* CRC, Gzip, Base64). This step corresponds to the contextual relationship inference (*c.f.* block 8.1 in figure 6.1) which produces, for each message, a contextual signature, *i.e.* an ordered sequence of types of contextual information.

Sometimes, some part of a given message can correspond to different contextual data. This situation is more frequent with short contextual data. Figure 6.5 illustrates such case where a single byte of the message is found as participating in the definition of several contextual information. In this example, the IP destination and the ID message are two types of contextual information found in the message. Indeed, the message ID can be found in two different places in the message: at the beginning of the message but also straddling the IP destination address.

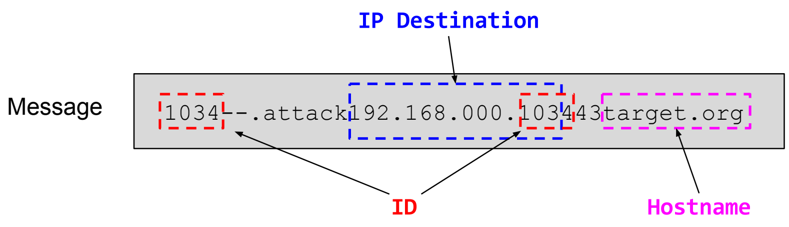


Figure 6.5 – Illustration of contextual conflicts.

To address this situation, we use two heuristics. The first applies when one contextual data is entirely embedded in the other. In that case, we give priority to the longest one. The second applies when the occurrences of two contextual data partially overlap. Our example falls in that case. To resolve this situation, we use a disjunction in the contextual signature. For example, the contextual signature of the previous message is `[ID;(IP Destination or ID);Hostname]` which accepts two valid realizations: `[ID;IP Destination;Hostname]` and `[ID;ID;Hostname]`.

We then use a greedy approach to cluster messages together if they share compatible contextual signatures, *i.e.* two signatures are said compatible if they are equal or if one is a possible realization of the other.

Finally, we compute the most precise common signature of each resulting cluster and iterate over its messages to tag their half-bytes. This last step consists in identifying each half-byte that correspond to each part of the contextual signature. In our first example message, we tag the eight first half-bytes with the tag “ID”, half-bytes between the 26th and the 60th half-bytes with both tags “ID” and “IP” while the last half-bytes are tagged with “HOSTNAME”. These tags are used by the format clustering to promote a semantic alignment between messages.

6.3.4 Format Clustering (Step 4)

Contextual clusters should be refined for two reasons: 1) to manage messages carrying no contextual information; and 2) to dissociate messages that include the same contextual information but have a different format. The format clustering step corresponds to the final stage of classification and is applied on each *contextual cluster*. Unlike the two previous steps, this clustering compares the alignment quality between messages to compute clusters.

We propose to extend both the Needleman & Wunsch (NW) [97] sequence alignment algorithm and the Unweighted Pair Group Method with Arithmetic mean (UPGMA) [127] hierarchical clustering algorithm. Our modifications take into account the semantic in both the alignment and the clustering phase. In the remainder, we give some details about these modifications.

Semantic Needleman& Wunsch

We first propose an extension of the NW algorithm to produce a semantic-aware common alignment between messages. In fact, NW can be applied on a symbol, which represents the common alignment of a set of messages. In the following, we use the term of message to both refer to messages and symbols. As described in Section 3.1.2, the original version of NW aligns two messages in two steps: 1) it fills a matrix with the similarity score of each pair of messages bytes and then 2) execute a back-trace in it. This matrix is filled accordingly to the principle of optimality described by formula (6.1). It uses a gap penalty d and a similarity function S to align messages m_1 and m_2.

$$F_{i,j} = max(F_{i-1,j-1} + S(m_1[i], m_2[j]), F_{i,j-1} + d, F_{i-1,j} + d) \tag{6.1}$$

In previous works [14, 43, 88], the similarity function S is reduced to a simple function $v(a, b)$ that either returns the value e if $a == b$ or f if not.

We propose to extend this syntactic comparison with the comparison of the semantic definition attached to each half-byte. Hence our function compares the value but also the semantic tags of each half-byte and preserves common semantic information if available. These semantic tags are computed and attached to half-bytes during the contextual clustering and every time an intra-symbol relationship is found.

We denote $\psi(a) = \langle T, \phi_a \rangle$, the multiset [132] of semantic tags attached to an half-byte a, with T the set of all semantic types and $\phi_a : T \to \mathbb{N}$, a function returning the multiplicity of a semantic tag in a. For example, $\psi(a) = \{\{IP, IP, Username\}\}$ means that IP and $Username$ semantic tags are attached to half-byte a. In this example the multiplicity of IP is two, *i.e.* $\phi(IP) = 2$. This situation may arise when the same semantic tag corresponds to different types of relationship. For example, an half-byte could correspond to both environmental and application information.

Now, suppose $\psi(a)$ and $\psi(b)$ respectively the multiset of semantic tags attached to half-byte a and b, we denote one includes the other with the relation:

$$\psi(a) \sqsubset \psi(b) \Leftrightarrow \forall e \in T, \phi_a(e) < \phi_b(e) \tag{6.2}$$

and we define a size function the following way:

$$\overline{\psi(a)} = \sum_{e \in T} \phi_a(e) \tag{6.3}$$

We compute the similarity between half-bytes a and b by comparing their values and their semantic tags. For the value comparison we keep the original $v(a, b)$ definition while for the semantic comparison we introduce two new semantic match and mismatch parameters: h and g. Our experimentation has shown best results with the following parameter values: $d = 0$, $e = 5$, $f = -5$, $g = 6e$ and $h = 6f$.

Hence, as described in table 6.1, our similarity function S returns a high score if the semantic tags match but the values differ and on the contrary, returns a low score if the values match but not the semantic tags.

$\psi(a) \cap \psi(b) = \oslash$	$S(a,b) = v(a,b) + h \times \overline{\psi(a)} + h \times \overline{\psi(b)}$
$\psi(a) = \psi(b)$	$S(a,b) = v(a,b) + g \times \overline{\psi(a)}$
$\psi(a) \sqsubset \psi(b)$	$S(a,b) = v(a,b) + g \times \overline{\psi(a)} + h \times \overline{\psi(b) \setminus \psi(a)}$
$\psi(a) \sqsupset \psi(b)$	$S(a,b) = v(a,b) + g \times \overline{\psi(b)} + h \times \overline{\psi(a) \setminus \psi(b)}$

Table 6.1 – Similarity function $S(a,b)$.

Once the matrix F is computed using our new similarity function and following formula 6.1, a trace-back step is performed. We rely on the original trace-back algorithm we described in Section 3.1.2. We search for a path that starts at $F_{|m_1|+1,|m_2|+1}$ and that maximizes the alignment score back to the origin $F_{1,1}$ [2]. A diagonal path describes a perfect alignment between the two messages, while a vertical or an horizontal motion implies the addition of gaps in one of the two messages. Such trace-back produces two messages m_1' and m_2' containing the necessary gaps to align messages m_1 and m_2 under the constraints introduced by their inner syntactic and semantic similarities.

As illustrated in figure 6.6, our semantic based alignment preserves the semantic definition when identifying token boundaries. In this example, without our solution, email addresses get split among multiple tokens and firstnames definition is lost in a bigger dynamic token.

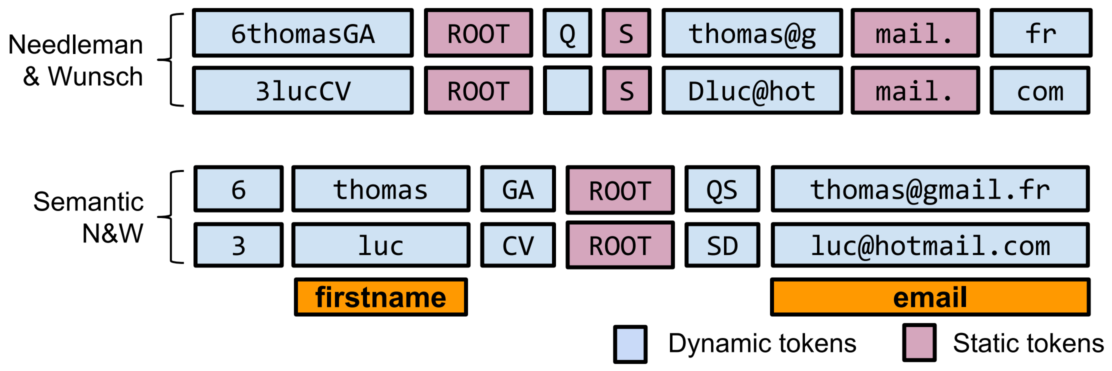


Figure 6.6 – Alignments computed by Needleman & Wunsch and of our modified version.

We leverage these two aligned messages to produce a symbol that describes both. As illustrated on Figure 6.7, our semantic NW alignment produces two aligned messages that may contain gaps. We build a symbol out of these messages by means of three steps: 1) we create a single representation of the aligned messages with a succession of static and dynamic tokens. 2) we smooth token boundaries and 3) finally compute fields definitions out of the smoothed tokens.

The objective of the first step is to find a succession of tokens that can describe the two aligned messages. To achieve this, we execute a pairwise comparison of each aligned message bytes. If both equals, we create a static token with its value, if not, we create a dynamic token to which we attach the two values. Once we compared all the bytes of the two aligned messages, we obtain a

2. $|m|$ denotes the number of half-bytes in a message m.

sequence of one-byte static and dynamic tokens as illustrated in Figure 6.7.

The second step smooths this sequence of one-byte tokens. To achieve this, we merge successive dynamic or static tokens that either share the same semantic or that have no semantic. This step produces a set of smoothed tokens as illustrated in Figure 6.7.

Finally, we create a symbol out of the sequence of smoothed tokens. In details, if multiple successive tokens participate in the same semantic definition we create a single field to represent them. A field is also created for each token that has no semantic definition. As described in Section 5.1.3, the values accepted by a field is represented under a token-tree. We therefore infer the token-tree of each field. If a field regroups multiple tokens, we represent them with an aggregate node (denoted `AGG` in Figure 6.7). We also infer the type of the values that are accepted by each token. If a token accepts a single value (*i.e.* a static token), we insert it in the token-tree of the field. On the other hand, we extract the types of the values that are accepted by each dynamic token. We rely on a heuristic that successively test if the values are of different types. We first test for strongly constrained types such as IPv4 addresses and then tests if the bytes are valid ASCII, or decimals. If all the bytes are valid printable characters we represent them as an ASCII sequence. If not and if the token is one, two or four bytes long, we represent its values under a decimal type. Other values are represented as a sequence of raw bytes.

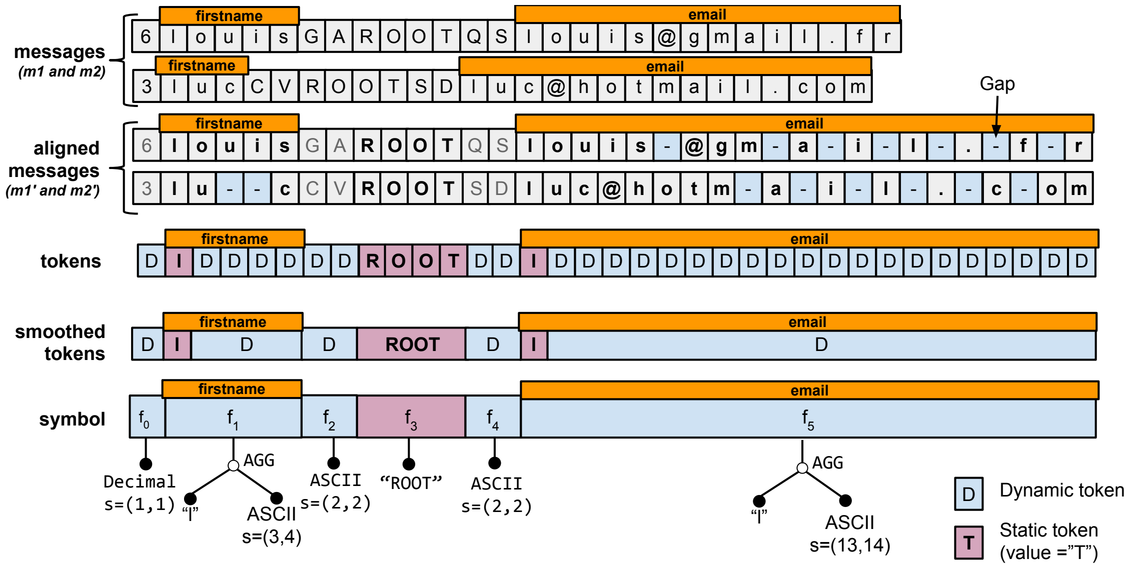


Figure 6.7 – The different steps engaged in the construction of a symbol out of two messages.

6.3.5 Semantic Preserving Clustering Algorithm (Step 5)

We use this extension of NW in our modification of the UPGMA algorithm. As explained in Section 3.1.2, UPGMA is a heuristic clustering algorithm that recursively joins the two nearest clusters. It relies on a matrix, denoting the pairwise similarity of clusters, *i.e.* of symbols.

In the n^{th} iteration of the algorithm, we try to align each pair of symbols resulting from the $(n-1)^{th}$ iteration using our modified NW algorithm. We also compute a new symbol for each pair of symbols and search for intra-symbol relationships. A semantic tag labeled with the type of the relationship is attached to each corresponding half-byte. More details on the relationship inference are provided in Section 6.4. After that, we compute the quality score of each possible merge using a dedicated function H. We finally merge the two symbols that maximize H into the new corresponding symbol provided by our modified NW algorithm. We stop this iterative merging process when the highest score falls below a specific threshold. Figure 6.8 illustrates how we regroup symbols with the UPGMA algorithm.

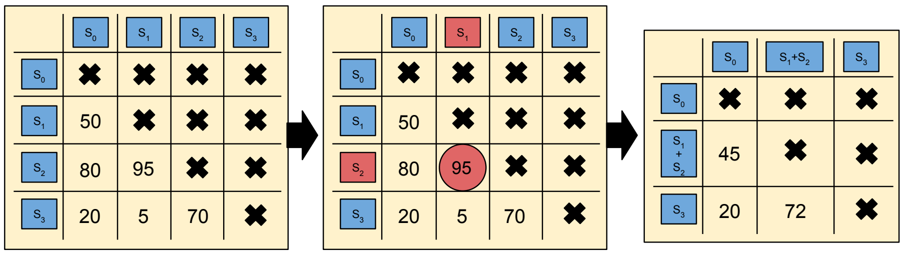


Figure 6.8 – Illustration of the UPGMA clustering algorithm.

Originally, after each merge, the UPGMA matrix is recomputed using an equation which estimates the similarity of the new clusters based on previous ones. However, because we wanted to keep track of the semantic definition when merging clusters, we modified this original behaviour. In our work, after each round we realign messages participating in the new clusters and recompute a symbol to represent it. We then compute its quality score. It allows us, for example, to detect that a semantic field will disappear if we merge two messages.

Hence, our UPGMA algorithm relies on $H(s_1, s_2)$. This function returns the quality score of s_{s_1,s_2}, a symbol representing the alignment of symbols s_1 and s_2. This H function computes the euclidean norm of a vector composed of two measures, $Q(s_1, s_2)$ and $P(s_1, s_2)$:

$$H(m_1, m_2) = \|Q(m_1, m_2), P(m_1, m_2)\| \tag{6.4}$$

The first measure $Q(s_1, s_2)$ represents the syntactic and semantic similarity of the two symbols and its value is extracted from the NW matrix F:

$$Q(m_1, m_2) = F_{|m_1|+1,|m_2|+1} \tag{6.5}$$

The second measure, denoted $P(s_1, s_2)$, evaluates the proportion of static half-bytes over the number of dynamic fields in the symbol resulting from the NW alignment of s_1 and s_2.

6.3.6 Merging Step & Inter-Symbol Relationship Identification (Steps 6 & 7)

Format clustering produces *Raw Symbols*, each denoting a possible message format. However, this approach tends to produce a lot of redundant *Raw Symbols* corresponding to the same message format. To address this issue, we added a simple merging step that compares all the computed *Raw Symbols* and merges duplicate ones.

The merging process successively compares the fields definition of each *Raw Symbols* and merge the ones that are equivalent. Two *Raw Symbols* are equivalent if they share the same sequence of fields with the same definition domains. When two *Raw Symbols* are found equivalent, we regroup all their messages under the same one and forget the other one.

Finally, the last step of our approach consists in identifying inter-symbol relationships. As detailed in section 6.4, it searches for relationships between consecutive messages using a generic relationship inference. The same approach is also used during previous step to find contextual and intra-symbol relationship inside each message. Thus we can identify fields such as sequence ID or cookies. This last step in our approach returns symbols representing different message format, including the definition of their fields.

6.4 Field Relationships Identification

In this section we present our approach to identify intra-symbol relationships (*i.e.* between fields that pertain to the same symbol) and inter-symbol relationships (*i.e.* between fields that pertain to two consecutive symbols). We consider three steps. At first, we generate a dataset that contains all combinations of field attribute couples. Then, we quickly eliminate bad candidates by means of a correlation approach, and finally we try to qualify the potential relationship that exists between the remaining field couples.

During the first step we generate a dataset with the following attributes for each field: its value, its size and its offset in current message. We then compute each possible combination of attributes couples as, for example, (field1.size, field3.value).

Then, we look in the dataset for correlations by leveraging the Maximal Information Coefficient (MIC) [112]. This coefficient retrieves many types of dependencies, including nonlinear ones. Moreover, it supports noisy datasets. This characteristic is useful as the clustering steps may have erroneously grouped messages of different formats, thus preventing us to find dependencies when looking for exact relationships. In order to differentiate linear from nonlinear dependencies (*e.g.* a size field from a CRC32 field), we combine the MIC score with r, the Pearson product-moment correlation coefficient. As demonstrated in [112], a value of $MIC - r^2$ close to zero indicates a linear dependency, whereas a score close to one tends to point out a nonlinear dependency.

The qualification step takes as input the best couple candidates considering their $MIC - r^2$ scores. We experimentally established that we obtain good results if we select couples between 0.8 and 1 for linear relationships and between 0 and 0.2 for non linear relationships. We do not consider scores between 0.2 and 0.8 as they generally lead to weak results in terms of relationships. We then evaluate each potential field couple under a set of specific relationships, in order to retrieve one that exactly applies, *i.e.* it should be valid for the entire set of messages.

In order to support the wide variety of encoding that exists in real protocols, we take into account different possible encoding and we try different combinations of endianness, signed number representation and byte interpretation (ASCII, decimal, hexadecimal and octal). In the current implementation, we consider the following basic relationships: size field, offset field, cookies and sequence number. We also consider the following complex relationships: SHA-1, CRC32 and the size of a repetition of a particular field or group of fields. The later can be found for example in the P2P ZeroAccess protocol, where a size field specifies the number of peer's IP address fields concatenated in a message.

Chapter 7

Comparative Study of Vocabulary Inference Approaches

We evaluate our approach on various protocols and compare our contributions against state-of-the-art approaches. We conducted two different types of experiments: 1) on well-known protocols to compare inferred message formats with their published specifications and 2) on unknown protocols to evaluate the effectiveness of the different approaches on more operational use cases.

For the first set of experiments, we selected a text protocol (FTP) and a binary protocol (SAMBA), both often used in previous experiments [44, 27, 41, 43]. The second set includes two typical use cases of protocol reverse engineering to cover more operational contexts: the P2P protocol used by a recent botnet known as ZeroAccess (ZA) [123] and Ventrilo [1], a proprietary and undocumented VoIP protocol.

In the remainder, we first give some key insights over the compared tools in Section 7.1, we describe the datasets in Section 7.2, the metrics in Section 7.3 and the implementations we used in the study in Section 7.4. We then conclude in Section 7.5 with a discussion on obtained results.

7.1 Choice of Compared Tools

As presented in Section 3.1, several previous works tried to tackle the problem of reverse engineering protocol vocabularies using trace-based approaches [88, 14, 43, 83, 139, 138, 82]. Unfortunately, no previous works have addressed their comparison. As a matter of fact, it is not easy to accurately determine the advantages and weaknesses of each approach. Two main reasons can explain this lack of comparative study. 1) Very few implementations of these works are available even for the scientific community [83] and 2) to our knowledge no datasets were published along with each work. Our comparative study tackles this issue.

As implementing these tools is time-consuming, we decided to retain the most representative ones while still covering the different types of approach. Thus, Discoverer [43] uses a syntactic alignment approach, ScriptGen [88] uses an inferred automaton and ASAP [83] relies on statistics

1. Ventrilo is a VoIP software: `http://www.ventrilo.com/`

over message bytes. We consider that other works use the same types of approaches and are less advanced or outdated by the tools we selected. Moreover, the selected tools are often cited in scientific articles of the domain [2]. Though ASAP is less popular, the tool is publicly available and other works [139, 82] that follow the same type of approach are very similar and not more popular.

ASAP, published by T. Krueger *et al.* in [83], focuses on message clustering. ASAP splits messages in $n - grams$ and searches among them for the most representative ones. To do so, they filter out keywords that have extreme (high and low) frequency of appearance. A Non-Negative Matrix Factorisation [86] is then performed to cluster similar messages. Finally, a template is extracted from each cluster to represent the message format associated with each cluster. However, the template is coarse-grained and not precise enough, especially to parse messages.

ScriptGen, developed by C. Leita *et al.* [88], includes features that both address the problem of vocabulary and the grammatical inference. It was initially designed to generate honeypot scripts [3]. ScriptGen differs from others because it uses the protocol automaton to identify similar messages. It passively builds an FSM by replaying the various sessions provided in traces. Messages that appear in the same state of the FSM are clustered together. Clusters are then subdivided following two main heuristics: 1) the number of bytes sent in response to a message; and 2) the result of Region Analysis algorithm execution. This algorithm is applied in two steps, first it clusters messages following a UPGMA execution coupled with a sequence alignment algorithm, then it subdivides obtained clusters following messages values.

Discoverer, by W. Cui *et al.* [43] tries to reverse unknown protocols following three main steps: tokenization, recursive clustering and merging. The tokenization process splits messages in ASCII and binary tokens to cluster messages that have the same token structure, *i.e.* the same sequence of token types. Then, the recursive clustering divides obtained clusters by identifying “format distinguisher” fields among them. To mitigate over-classification problems, the last step merges similar message formats by using a type-based sequence alignment.

7.2 Datasets

Our comparative study relies on six datasets: two of them (① and ②) correspond to a well-known text protocol (FTP), two of them (③ and ④) to the well-known SAMBAv2 binary protocol (SMB), one to a P2P botnet protocol (⑤) and one to a typical commercial proprietary product (⑥). Table 7.1 summarises the different dataset characteristics.

To compare the best results of each tool, we use two kind of datasets for each known protocol (FTP and SMB): a calibration dataset to empirically compute the optimal parameters of each tool and an evaluation dataset to compare them.

To create the calibration datasets ① and ③, we used the first solution detailed in section 6.2 to create scripts that execute various actions with predefined parameters on the protocol implementation. For instance, the FTP script executes more than 10 different actions, including a connection attempt with a bad password, listing some directories and downloading multiple files. Each cali-

2. According to Google Scholar, ScriptGen is cited 123 times, Discoverer is cited 150 times, ASAP is cited 10 times.
3. See Honeyd project: `http://www.honeyd.org/`

#	Protocol	Source	# Msg	# True Format
①	FTP	Generated	1717	40
②		LBNL	2328	46
③	SMB	Generated	2650	32
④		Company	937	22
⑤	ZAccess	Public	883	4
⑥	Commercial	Laboratory	482	15

Table 7.1 – Summary of datasets used in our comparative study. The first column denotes the dataset identifier. Last columns denote the number of true formats and the number of messages in the dataset.

bration dataset includes twenty application sessions containing the same actions but with different contextual parameters, *ie.* usernames, filenames, IP addresses and hostnames. Thus, we annotated the captured traces with the executed actions and contextual data used to generate them.

To create the evaluation datasets, we used traces captured in both academic and professional environments. The realistic FTP dataset (②) is a subset of traces published by LBNL [104], collected in an university network. We arbitrary considered the first 1000 packets in three different days of capture (days 10, 11 and 12) to produce a dataset of reasonable size. The second realistic dataset (④) comes from a full day of SMB traffic captured in a company network. Users agreed to participate and behaved in a normal way. We retained a portion of the whole traffic that represents 1000 packets. Obtained dataset is composed of 937 distinct SMB packets, covering 22 different true formats. By true format, we hereby refer to the format detailed in protocol specifications.

For anonymity reasons, the LBNL dataset only includes traces that hold no precise definition of the context in which they were captured. In such situation, we would have used the last solution proposed in Section 6.2 to obtain necessary semantic information. However, in that case this datasets would not reflect the same quality as those used for calibration. Returned results would therefore be difficult to interpret as various factors would have influenced them. Thus, to ensure consistency between parameters used for calibration and evaluation, we extracted from evaluation network traces the same types of contextual data than the one we used for calibration. We relied on the Wireshark tool that can be use to extract the contextual information we were looking for. We followed the same approach on the SMB datasets.

Finally, we applied the four approaches on more realistic reverse engineering situations: i) the P2P communication protocol used by ZeroAccess [123] botnet and ii) a subset of the protocol used by a commercial VoIP product. To create the dataset of ZeroAccess traces (⑤), we used the second version of the malware, provided by K. McNamee [94]. We deployed this malware in a confined and controlled network infrastructure. We then allowed our sample to connect with other botnet members through its P2P protocol (used to retrieve the P2P directory). To capture the traffic, we used a network probe implementing the deobfuscation algorithm previously detailed by K. McNamee [94]. The obtained dataset includes 883 messages for four true formats, *i.e.* we previously performed a manual reverse engineering of the protocol to identify its true formats.

Regarding contextual data, we extracted various information from the pcaps meta-data such as IP addresses and port numbers. Even though it does no bring a lot of contextual data, it still provides good results as messages generated by this P2P protocol often include network related information, such as the IP addresses of its peers. For the last dataset (⑥), we considered the protocol of a typical VoIP commercial product. To obtain traces of this protocol, we relied on a freely available implementation of this protocol [4] and automatised its execution. In accordance with the solution we detailed in Section 6.2, we selected a subset of the application features and established a scenario of 10 different actions, such as sending text messages, configuring personal data, disconnecting from or connecting to the server. We also arbitrary defined the values of each action parameters and stored them as contextual data. We then played this scenario three times and captured the generated traffic to create a dataset of 482 messages.

To ensure the reproducibility of these experiments, we stored and archived these datasets, but only some of them are made public. Generated datasets (① ③), FTP realistic dataset (②) and ZA dataset (⑤) are available for download. However, the realistic dataset of SMB packets captured from a company network cannot be published due to embedded sensitive information, and the one extracted from the commercial product cannot be published due to intellectual property restrictions. We also archived all the identified contextual information for each protocol that we summarised in table 7.2.

7.3 Metrics

To measure and compare the effectiveness of message formats inference algorithms, we need to define metrics. We reviewed all the metrics used in previous works experiments [43, 41, 139, 9, 29, 89] but no consensus emerged. For instance, some works report similar metrics but with different names (*i.e.* [43] and [41]), some compute their own measures [139, 9] and others only use qualitative metrics in their experiments [29, 89]. We select two metrics and propose a new one to cover our needs in the evaluation of message clustering and field partitioning: the correctness, the conciseness and the precision. Correctness and conciseness are both used in [43] and are closed to the metrics used in [89, 41]. Figure 7.1 illustrates the three metrics we use.

To define the conciseness and the correctness of a clustering algorithm we consider M a set of messages, F_{inf} the set of inferred formats, *i.e* of inferred symbols, and F_{true} the set of true formats, *i.e.* symbols defined in the protocol specification. We also denote the function $I : M \rightarrow F_{inf}$, which defines the inferred format of a message and the function $T : M \rightarrow F_{true}$, which defines the true format of a message. We finally define two functions, $N_{con} : F_{true} \times M \rightarrow \mathbb{N}$ and $N_{cor} : F_{inferred} \times M \rightarrow \mathbb{N}$:

$$N_{con}(f, M) = |\{I(m),\ \forall m \in M \text{ such that } T(m) = f\}| \tag{7.1}$$

$$N_{cor}(f, M) = |\{T(m),\ \forall m \in M \text{ such that } I(m) = f\}| \tag{7.2}$$

4. Client Ventrilo for Windows is available at `http://www.ventrilo.com/download.php`

Protocols	Identified Actions	Identified Contextual Information
FTP	Connecting with a bad password, connection with a valid password, listing current directory, moving in an invalid directory, moving in a valid directory, downloading a file, uploading a file, closing the connection	Client username, client password, server hostname, current directory, downloaded filename, uploaded filename, name of the moving directory, invalid directory name, listed directories.
SMB	Connecting with a bad password, connecting with a valid password, listing available shares, moving in an invalid directory, listing a directory, downloading a file, uploading a file, closing the connection.	Client username, client password, server hostname, downloaded filenames, uploaded filenames, moving directory names, listed filenames, server domain name, server os, server version, server shares.
ZeroAccess	Receiving a new peer address, propagating a peer list.	IP addresses and UDP ports found in the pcap.
Commercial VoIP	Connection client 1, connecting client, client 1 sends a message to client 2, client 2 sends a message to client 1, client 1 changes its configuration, client 2 changes its configuration, client 1 disconnects, client 2 disconnects.	Server IP, server hostname, client IP, client phonetic names (a parameter of the client configuration), client description, client comment message, client comment url, client messages.

Table 7.2 – Identified Semantics.

Those two metrics are related to the mapping between true formats and inferred formats. Intuitively, the clustering is correct if it computes homogeneous clusters. It means that every cluster, *i.e.* inferred format, must contain only messages that share the same true format. In this case $N_{cor}(f) = 1$. Heterogeneous clusters decrease the correctness and in this case $N_{cor}(f) > 1$. Conversely, the clustering is concise if each true format is described by at most one inferred format. When messages corresponding to the same true format are clustered into different inferred formats, conciseness decrease and $N_{con}(f) > 1$. The clustering is correct if $N_{cor}(f)$ remains low (ideally equal to one) for a large number of true formats. It is concise if $N_{con}(f)$ remains low (ideally equal to one) for a large number of inferred formats. We thus define Conciseness (respectively Correctness) as the Cumulative Distributed Function (CDF) of N_{con} (respectively N_{corr}): $Con(n) = p(N_{con} <= n)$ and $Cor(n) = p(N_{cor} <= n)$ with $p(x <= n)$ the probability that $x <= n$.

The overall shape of such CDF curves characterizes the correctness or the conciseness of a given clustering approach. However, two points of such curves are of particular interests. The first one corresponds to $Cor(1) = p(N_{cor} = 1)$, *i.e.* the proportion of homogeneous clusters and the second one to $Con(1) = p(N_{con} = 1)$, *i.e.* the proportion of true format that correspond to at most

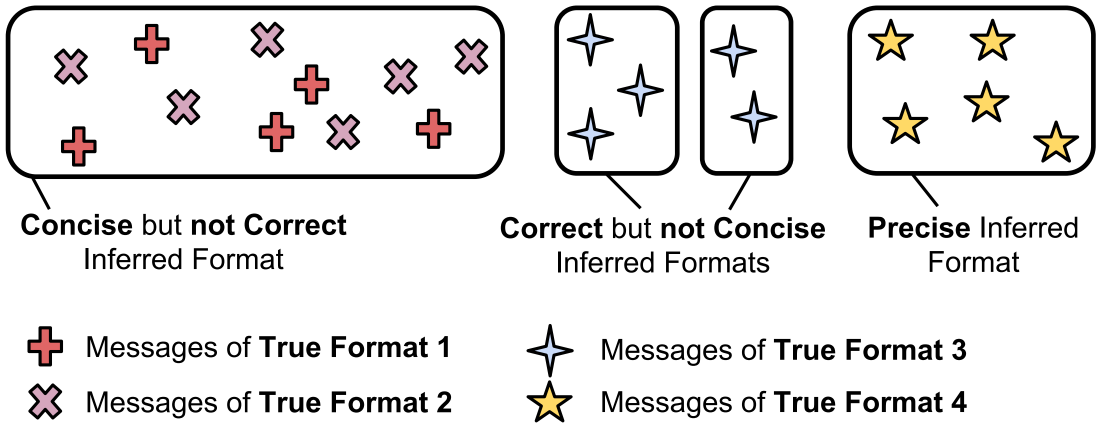

Concise but not Correct Inferred Format
Correct but not Concise Inferred Formats
Precise Inferred Format
Messages of True Format 1
Messages of True Format 2
Messages of True Format 3
Messages of True Format 4

Figure 7.1 – Illustration of our three metrics: conciseness, correctness and precision.

one inferred format. We thus expect such values to be high.

Correctness and conciseness must be analysed together to understand the quality of the alignment. For example, an inference algorithm that classifies every messages into different clusters will be described as very accurate but unconcise. For this reason, we also use Receiver Operating Characteristic (ROC) curves to compare the different tools. We compute these curves based on the values of $Corr(1)$ and $Con(1)$.

However, we believe these metrics are not sufficient to gain a precise vision of the overall quality of the inference processes. To improve this, we propose an additional metric that focuses on the precision of the classification. To measure it, we compute the number of inferred formats that perfectly match a true format. We call them precise formats or precise clusters.

To use these metrics, we need to known the true format associated with each message. To identify the true format of well-known protocols, such as FTP and SMB, we used the results of Wireshark. More precisely, we extract the needed information out of a PDML file generated by Wireshark [5]. This file includes the description of all the fields of the captured messages. We then use a protocol-specific PDML parser that exposes the values of some key fields embedded in each message. We assume that all the messages that share the same key values correspond to the same true format. To select these key fields we refer to the official specifications of the protocols and only select important fields. For instance, for the SMB protocol we consider the value of the "SMB Command" field, the value of optional "subCommands" fields and the value of the "status" field. Unlike previous work [43], we believe optional fields do not participate in the definition of a true format. For the two other protocols, we manually create specific parsers based on previous works [123] [6] published by other researchers that reversed such proprietary protocols.

5. Wireshark is a famous free network capture tool: `http://www.wireshark.com`
6. Project Mangler as revealed most parts of the Ventrilo protocol: `http://www.mangler.org/`

7.4 Implementations

As explained previously, this comparative study relies on our re-implementation of retained works. This section details them. We first present our framework and then give relevant implementation details of our approach as well as on our re-implementation of ASAP, ScriptGen and Discoverer.

Our open source framework for reverse engineering of communication protocols [7] is licensed under GPLv3. Freely available, its sources can be downloaded from a git repository and some packages for Linux platforms are provided. At the time of writing (October 2014), the source code of the framework comprises more than 50,000 lines of code, mostly in Python, some specific parts being implemented in C for performance purpose. It offers data models and basic algorithms to build, edit, visualise and simulate a communication protocol. We therefore implemented our approach and the others as plugins to reduce duplicated code and to simplify their comparisons.

The implementation of our approach of vocabulary reverse engineering corresponds to only 500 lines of python since most of the computation codes are provided by our framework. We refer to it as Netzob in the following.

ASAP authors provide a publicly available implementation in R [8]. Thanks to the help of the authors, we developed a wrapper to execute ASAP implementation as a clustering plugin in our framework. This allows us to use the original ASAP code without inserting flaws. As recommended by authors, we uses Sally [115] to tokenize messages. Output clusters returned by ASAP are then transformed into symbols as presented in their article [83].

Unfortunately, implementing ScriptGen and Discoverer was much more difficult since neither source code nor implementation are publicly available, even for the scientific community. Among the two, Discoverer was the most difficult to re-implement as documentations and articles give too few details on some specific points such as on their merging strategy used in last step. In addition, both the authors of Discoverer and ScripGen did not publish the dataset they used to evaluate the effectiveness of their tools. Without publicly available datasets, it is thus difficult to validate our implementation of these approaches. However, we try to be as accurate as possible and check carefully our implementations of these approaches. Moreover, the results we obtained are similar to those described in the authors articles.

Each approach exposes parameters to the user that can highly impact the overall quality of their results when applied on a certain type of protocol. Thus, to ensure a fair comparison, we use the calibration datasets to compute the best parameters value and use them on their respective evaluation datasets. To identify these parameters, listed in Table 7.3, we first established a large variation range for each of them and then compared the ROC curve of results brought by all possible combinations. For realistic datasets with no calibration, we only retained the best results given all the possible combination of parameter values.

7. Netzob- Reverse Engineering Communication Protocols: `http://www.netzob.org`
8. ASAP sources: `https://github.com/tammok/PRISMA/`

Tools	Parameters	FTP	SMB	ZA	Commercial VoIP
ASAP	Ngram Length	1	2	1	1
	Ngram Type	Text	Bin	Bin	Bin
	Ngram Delimiters	Extended	-	-	-
Discoverer	Min. Text Segments	2	2	2	2
	Min. Cluster Size	20	60	10	12
	Max. Distinct Values	10	5	20	350
ScriptGen	Macro-Clustering threshold	0.9	0.7	0.6	0.6
	Micro-Clustering threshold	0.5	0.5	0.4	0.4
Netzob	Similarity Score	0.9	0.6	0.5	0.8
	UPGMA threshold	10	10	10	-1

Table 7.3 – Parameters used to configure each approach.

7.5 Experimental Results

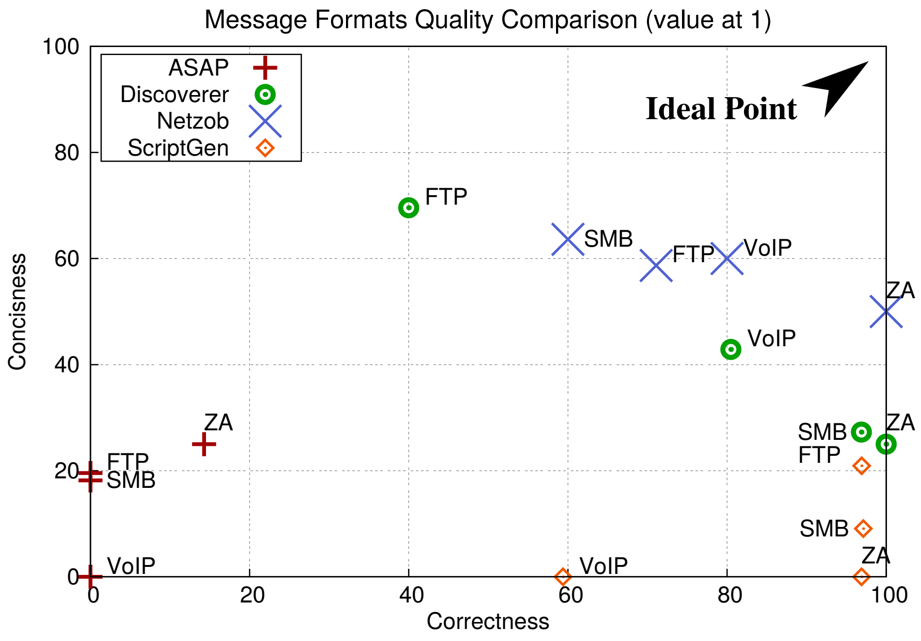


Figure 7.2 – ROC Curve used to compare the quality of the inferred message clusters of ASAP, ScriptGen, Discoverer and Netzob. Best results are close to the top right corner.

In this section, we present the conclusions of our experimental comparative study of ASAP, ScriptGen, Discoverer and Netzob.

We expect results to be both concise and correct. This means that the ideal point of our ROC curves is the upper right corner of the graph, as illustrated in figure 7.2. It is also important that the tools balance concision with correctness. Conversely, a lost in conciseness entails an important problem: it generates too many symbols, which make the results difficult to interpret. Moreover, these symbols can be used to infer the grammar of the protocol and then to develop protocol generators. The size of the inferred protocol grammar automaton depends on the number of inferred symbols. A lost in conciseness will thus result in inefficient protocol generators which may prevent any inference of the protocol grammar. Concerning the ROC curve, this means that good results should be as close as possible to the upper right corner, *i.e.* the "Ideal Point" on figure 7.2, and near the diagonal that goes from origin to that upper right corner.

First of all, general results depicted in figure 7.2 show that the compared approaches fall into two categories. On one hand, ASAP and ScriptGen obtained poor results and always suffer from the overfitting problem. Moreover, ASAP correctness is quite low meaning that most of its inferred message formats denotes multiple true formats. On the other hand, Discoverer and Netzob show better results with an advantage for Netzob which results are always nearer the ideal point.

The precision of the clustering illustrated in Table 7.4 is also another revealing measure of this distinction. Indeed, only Discoverer and Netzob infer precise clusters, *i.e.* inferred clusters that perfectly match true formats. Despite the use of calibration datasets to optimize their parameters value, ScriptGen and ASAP inferred clusters never matched a true format. Indeed, none of their inferred message formats is accurate enough to support the automatic generation of a protocol parser.

Precision	FTP (②)	SMB (④)	ZA (⑤)	VoIP (⑥)
Discoverer	4.34%	22.72%	25%	6.66%
Netzob	**34.78%**	22.72%	**50%**	**26.6%**

Table 7.4 – Number of precise clusters identified by Discoverer and Netzob.

We also observe in figure 7.2 that ASAP, Netzob and ScriptGen tend to be stable as they provide similar results for the different datasets. However, Discoverer is quite unstable. On SMB and ZA datasets, it suffers from the overfitting problem whereas on the FTP dataset it obtains a good conciseness but lower correctness (about 40%).

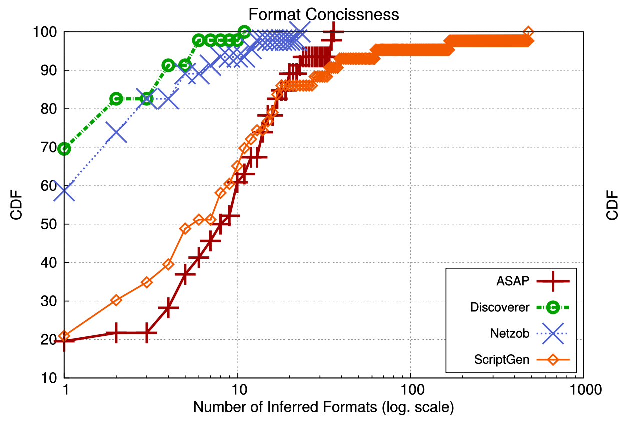
Our comparative study shows that ASAP does not return good results on the datasets we used. For instance, applied on the FTP realistic dataset (②), only 20% of the true formats match a unique inferred format (*c.f.* figure 7.2). We believe ASAP is not appropriate to infer precise specifications of a protocol. An approach solely based on a statistical analysis of keyword or n-grams in messages, does not appear sufficient to cluster them precisely.

Another interesting point in our results is that ScriptGen creates far too much clusters. For instance, on the SMB realistic dataset (④) made of 937 messages for 22 true formats, ScriptGen infers 906 different message formats. Indeed, most of the computed clusters contain a single message. Figure 7.3(c) depicts this low conciseness problem when applied on SMB dataset: it needs more than 100 inferred formats to cover 100% of the true formats. The reason why ScriptGen over-classifies is that it clusters messages according to their position in a session. This is not efficient, because in realistic datasets users often behave differently in each session which brings different message formats at similar position in sessions. Besides, when a classification error occurs at the beginning of the session, it affects the classification quality of the all following messages.

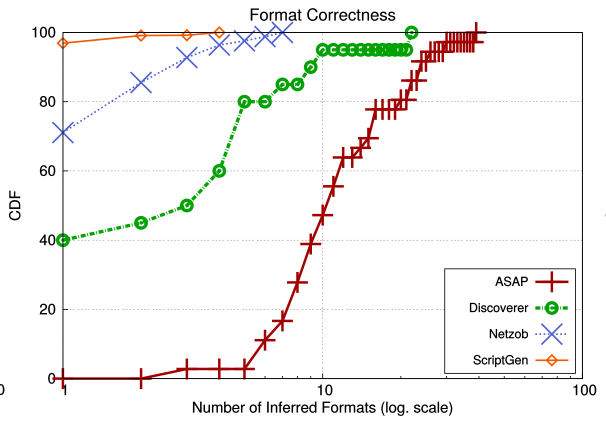
Obtained results confirm that ScriptGen was not designed to achieve a complete reverse engineering of a protocol. Indeed, it seems more appropriate for the inference of the very first exchanges of a communication. As stated by its authors, ScriptGen is more adapted to build an agnostic honeypot.

As illustrated in figure 7.2, our comparative study shows that among all the tools, Netzob always infers the best message formats. Indeed, it always computes message formats with a minimum correctness of 60% and a minimum conciseness of 50% whereas Discoverer lack of conciseness on binary protocols produces hundred of inferred message formats for a single true format.

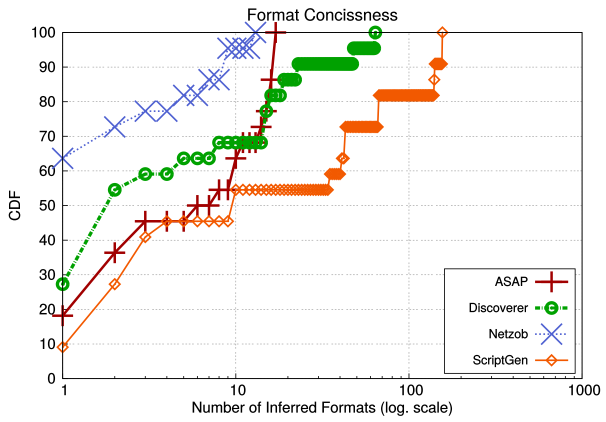
In addition to its stability, Netzob also computes the highest rate of precise clusters. Table 7.4 shows that Netzob always infers at least (and often more) precise clusters than Discoverer. From our point of view, inferring precise clusters is important. Not only that precise clusters represents perfectly inferred format but they can also help the expert to correct the other inferred message formats. That is because in most of the protocols, the different messages shares common aspects such as encoding functions or delimiters. The expert can apply this information on other inferred format to improve them.



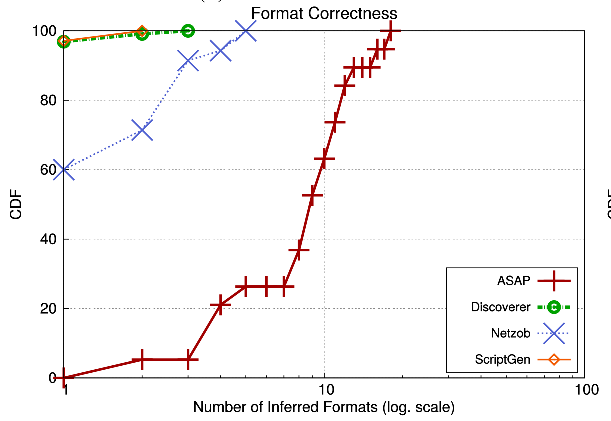
(a) FTP Concissness



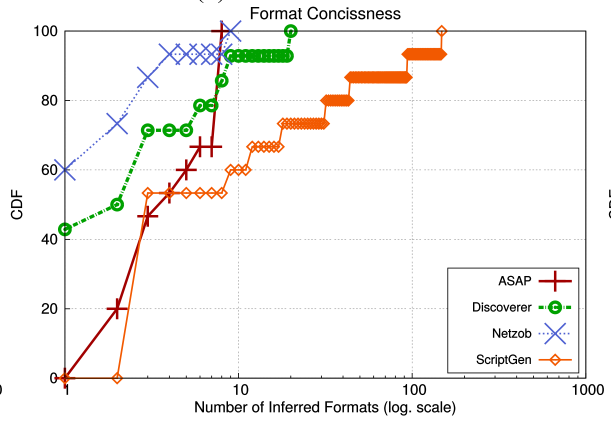
(b) FTP Correctness



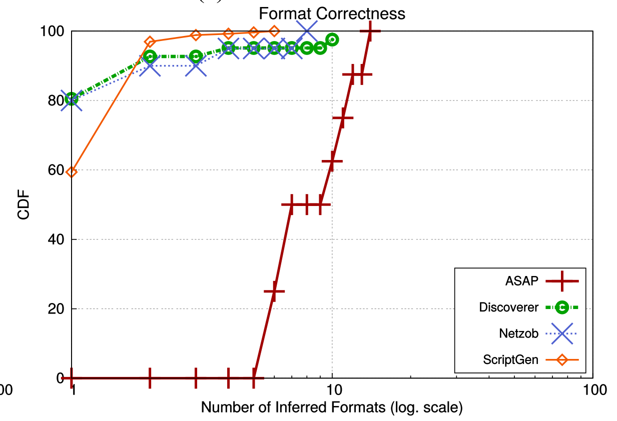
(c) SMB Concissness



(d) SMB Correctness



(e) Commercial VoIP Concissness



(f) Commercial VoIP Correctness

Figure 7.3 – Detailed Experimental Results

Chapter 8

Conclusion on Vocabulary Inference

In this Chapter, we proposed a complete and automated approach for trace-based message formats reverse engineering. Our approach relies on novel techniques that leverage contextual information and correlation means to enhance message clustering as well as field boundaries and relationship identification. We implemented our approach in a publicly available framework, and demonstrated its efficiency against both standard and unknown protocols. Moreover, we compared our approach against three other state-of-the-art approaches (Discoverer, ASAP and ScriptGen). The experimentation shows that it provides better overall results, in addition to extracting fields semantic.

Based on these results, network security products editors can rely on an approach that automates the creation of protocol parsers, thus providing fast and reactive response adapted to today's cybersecurity context. Same goes with the field of malware analysis, in the aim of speeding up, for example, the take down of botnets. Besides, the inferred protocol vocabulary can be also be use to tackle the inference of the protocol grammar. We detail our work in the field of protocol reverse engineering in next Part.

Part II

Automated Inference of the Protocol Grammar

Chapter 9

Introduction

We described in Part I our work in the field of vocabulary protocol inference. This work has lead to our proposition of an automated approach that leverages semantic information to reverse engineer the vocabulary of an documented protocol. Once inferred, the vocabulary describes the set of messages that are accepted by the targeted protocol. However, it does not specify the valid sequences of messages the protocol accepts, an information that is modeled by the grammar of the protocol. We explained in Chapter 1.2 that this knowledge is mandatory for the creation of realistic traffic generators, IDS detection rules and smart fuzzers. We therefore extended our work to propose an automatic approach to infer the grammar of a protocol.

We detailed in section 3.2 that previous works [9, 139, 124, 41, 67, 20, 35] have already applied the field of grammatical inference to the particular aspects of protocol RE. Our analysis of these studies and of the completeness and correctness of the inferred grammar they provide encouraged us to adopt an active inference approach. However, we also explained in Section 3.2.2 the main limitations that need to be faced to ensure the adoption of active grammatical inference approaches by the particular field of security related researches. Among these limitations, we refer to the important computation times these works require when applied on complex protocols. We also highlighted in this Section the need in more a stealthy process to address the inference of protected implementations. We believe that semantic information can also be a key parameter to address these issues. Indeed, we show in this Part that semantic information can be leveraged to split the large inference task into separate parallel sub-tasks. Our solution reduces the computation time of the whole inference and the stimulation of the inferred implementation thus being more stealthy.

Similarly to previous work [20, 35], we rely on the state-of-the-art inference algorithm called L^* that applies on protocols modeled with a DFA. However, some communication protocols are far more complex and cannot be modeled with DFAs. For example, the routing protocol BGP is an example of a Turing-complete protocol [34]. Indeed, we believe that communication protocols are closed to programming languages which for some of them are Turing-complete. Inferring such languages and the models they rely on (*e.g.* linear-bounded non-deterministic automaton, Turing machine) is a very complex work that has not yet been fully addressed by the scientific community. As a matter of facts, grammatical inference algorithms are still limited to the first levels of these languages which mostly relies on deterministic automaton. Our work do not derogate from this rule. Thus, we mostly focus on the inference of regular languages. Nonetheless, our use of a Mealy machine combined with a symbolic vocabulary that enables some context sensitivity through its memorization strategy (SVAS) allows us to address more complex protocols such as the ones that includes inter-message relationships.

In the sequel, Chapter 10 describes our model of a protocol grammar and details how we plan to decompose it to improve its inference. Chapter 11 describes our solution to infer this model. We then conclude with an evaluation of our solution we compare against the classical L^* algorithm in Chapter 12.

Chapter 10

Our model of a Protocol Grammar

As stated by Holzman, the grammar in a communication protocol represents the valid sequences of received and emitted messages. The automaton theory, as being closely related to formal language theory, is adapted to model rules that represent these sequences. Among all the existing models of automaton, *Finite-State Machines* (FSMs) with outputs and more precisely *Mealy machines* have successfully been used in previous works [35, 2, 17]. In the following, we describe our Symbolic Mealy Machine in section 10.1. We detail in Section 10.2 how our model support the reaction time to improve the realism of the generated traffic. Finally, Section 10.3 details our decomposition of this grammar model into sub-grammars to improve the efficiency of its inference.

10.1 Symbolic Mealy Machine

Definition A *Mealy* machine $\mathcal{M}$ is defined by a tuple $\langle Q, q_0, \Sigma'_I, \Sigma'_O, \delta, \lambda\rangle$ where Q is a nonempty set of *states*, $q_0 \in Q$ the initial state, Σ'_I and Σ'_O respectively the input and output alphabets, $\delta : Q \times \Sigma'_I \to Q$ the *transition* function and $\lambda : Q \times \Sigma'_I \to \Sigma_O$ the *output* function. The *transition* function defines the modification of the current state given an input symbol $a \in \Sigma'_I$. The *output* function models the transmission of an output symbol $b \in \Sigma'_O$ given the current state and the input message.

We use the notation $\textcircled{q} \xrightarrow{a/b} \textcircled{q}'$, proposed by F. Aarts [2], to represent the transition in $\mathcal{M}$ from state $q \in Q$ to $q' \in Q$ triggered by the reception of symbol $a \in \Sigma_I$ and the transmission of symbol $b \in \Sigma_O$ in response. Thus $\textcircled{q} \xrightarrow{a/b} \textcircled{q}'$ denotes $\delta(q, a) = q'$ and $\lambda(q, a) = b$. Figure 10.1 illustrates a Mealy machine that models a simple communication protocol. This machine is composed of three states, an initial state (0) and six transitions including transition $\textcircled{1} \xrightarrow{\texttt{Pass/Ack}} \textcircled{2}$.

For sake of readability, we do not explicit every transitions and propose instead a default behavior for the unspecified ones. To formalize this, we introduce $\theta : Q \to \mathscr{P}(\Sigma'_I)$, a function that returns the list of input symbols that trigger the execution of an explicitly defined transition in a given state. For example, applied on the automaton described in figure 10.1, $\theta(0) = \{\texttt{Hello}\}$ and $\theta(1) = \{\texttt{Whoami?}, \texttt{Exit}, \texttt{Pass}\}$. If for a state $q \in Q$ and a given input symbol $a \in \Sigma'_I$ a transition is not explicitly defined, *i.e.* $a \notin \theta(q)$, it implicitly means that a self loop transition exists

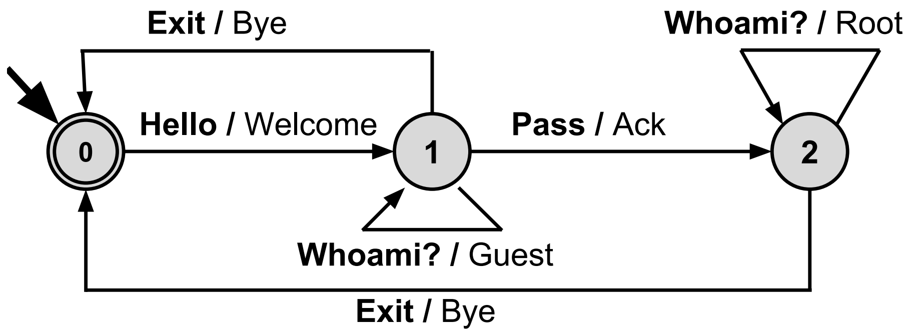


Figure 10.1 – Example of a simple protocol modeled as a Mealy machine.

such that $\delta(q, a) = q$. The execution of such loop triggers the emission of an empty output symbol $\epsilon \in \Sigma'_O$: $\lambda(q, a) = \epsilon$. This *empty symbol* represents the absence of symbol emission or reception during a given period of time, ω, which is a parameter of our model.

10.2 Reaction Time

As detailed in Section 1.3, one of the objectives of our model is to build a realistic traffic generator that can be inferred out of unknown protocols. To achieve this, we also model the reaction time. Indeed, this timing information increases the generated traffic realism by representing the computation time required for an implementation to parse a message, execute the requested operations and emit an answer. We therefore attach a reaction time to each transition of the grammar of a protocol.

This reaction time may vary due to a lot of factors over which we have no control. Among them, some properties of the environment such as the physical distance between actors of the communication, the available bandwidth or the physical equipment used to support the communications. Because of all these factors, we model the reaction time following a normal distribution assuming we have no previous knowledge on its definition. Formally, the reaction time is modeled by function $\Omega : Q \times \Sigma_I \times \Sigma_O \to \mathbb{N} \times \mathbb{N}$ that represents the mean and the standard deviation of the reaction time. The transition notation is therefore extended to support it, thus $⓪ \xrightarrow[\mu_b, \sigma_b]{a/b} ①$, denotes a transition between states ⓪ and ① triggered by the reception of input symbol a and the emission of output symbol b after a reaction time modeled by $\mathcal{N}\{\mu_b, \sigma_b\}$. To be consistent with our definition of an empty symbol, we define $\forall q \in Q, \forall a \in \theta(q), \Omega(q, a, \epsilon) = \mathcal{N}\{w, 0\}$.

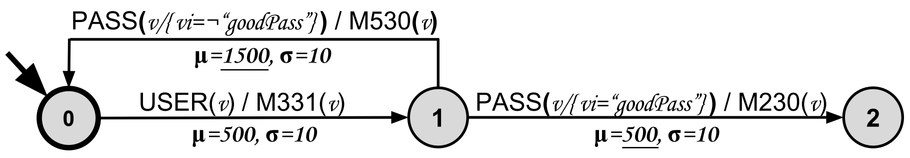

PASS(v/{vi=¬"goodPass"}) / M530(v)
μ=1500, σ=10
USER(v) / M331(v)
μ=500, σ=10
PASS(v/{vi="goodPass"}) / M230(v)
μ=500, σ=10
0
1
2

Figure 10.2 – Model of the FTP authentication schema with time definition.

Applied to the FTP authentication example, we can specify that a failed login takes more time to compute than a successful authentication as illustrated on figure 10.2.

10.3 Decomposing the Protocol Model to Improve its Inference

The L^* algorithm can be used to infer a complete and concise DFA out of a targeted protocol. However, as explained in section 3.2.2, the required number of queries, the inference time and the non-stealthiness of this approach can prevent its usage on protocols modeled by state machine that have numerous states. To address these issues, we propose to independently infer sub-parts of the protocol grammar before merging them to obtain the whole grammar of the protocol.

Our approach relies on the assumption that the grammar of a protocol can be decomposed into several simpler components, we call sub-grammars. As noted by H. Zafar [61], decomposing complex automata into simpler components has been the subject of numerous works [11, 45, 60, 10]. Indeed, these methods are of particular interest to optimize the synthesis of FSMs used in Field Programmable Gate Arrays (FPGA) and Programmable Logic Devices (PLD). In our work, we seek to leverage this concept of FSM decomposition to optimize the reverse engineering of the grammar of a communication protocol. Applied to our field of interest, inferring parts of the grammar before merging them has multiple advantages. We detail them in the following.

If sub-grammars have fewer states and smaller alphabets than the whole protocol grammar, their inference require less queries. Indeed, we stated in Section 3.2.2 that the theoretical upper bound of the number of queries in L^* is: $\mathcal{O}(|\Sigma|mn^2)$ where $|\Sigma|$ is the size of the input alphabet, m the maximum length of the counter-example and n the number of states of the inferred state machine. In our case, we execute multiple instances of L^* each inferring a small part of the grammar. Our approach has an upper bound complexity of $\mathcal{O}(p\overline{|\Sigma_a|}\overline{m}_a\overline{n}_a^2)$ with p the number of sub-grammars, $\overline{\Sigma}_a$ the average number of symbols per sub-grammar, $\overline{m}_a$ the average longest counter-example per sub-grammar and $\overline{n}_a$ the average number of states per sub-grammar. Thus, by reducing the number of states over which the algorithm applies, *i.e.* $\overline{n_a} \ll n$, we significantly reduce the value of the preponderant variable in the overall complexity.

In addition, by breaking the protocol grammar in sub-grammars, we can parallelize the execution of our algorithm. In this case, the total inference time to infer the grammar is similar to the inference time of the largest sub-grammar. If we can decompose the grammar in small sub-grammars, it highly reduces the inference time.

Another advantage is that our approach is more stealthy. Indeed, we can observe that the different symbols often satisfy the principle of locality. This means that a given symbol is often used only in a subpart of the protocol, *i.e.* of the automata. For example, in the FTP protocol, symbols `LOGIN` and `PASS` only participate in the authentication phase. Using such symbol during any other phase will result in invalid sequences. By default, the L^* algorithm does not take this principle of locality into account. Indeed, the use of L^* may generate thousands of protocol errors increasing the risk of being detected during the inference process. With our approach, we only use a subpart of the vocabulary to infer each sub-grammar. We therefore leverage the principle of locality thus reducing the probability of emitting a message not related to the current phase.

Finally, two more advantages arise with our approach: 1) it supports the exclusion of a portion of the protocol grammar from the inference process and 2) enables an incremental inference process. The benefits of the first advantage especially appears when a part of the protocol is protected by a security mechanism, for example when the user is automatically banned when he makes a mistake during its authentication. Our solution can therefore be useful to overcome such issue by excluding the protected part from the inference while still learning the other parts. Besides, our solution also supports the incremental inference of the protocol grammar. Indeed, our approach supports the extension of an inferred protocol grammar with an additional sub-part of it, we did not inferred previously. Our approach allows this without re-executing the entire inference process.

Chapter 11

Learning the Grammar Using an FSM Decomposition

Learning the grammar of a communication protocol consists in inferring the rules that define the valid sequences of sent and received symbols. As detailed in section 3.2, previous work addressing this issue can be divided in two families: passive and active approaches. Passive algorithms are faster and much more simple to implement but resulting automata often lack in completeness and can be erroneous. Active inference algorithms compute more complete and concise results but are notably slower. We propose an hybrid approach that combines a passive and an active approach. Our objective is to reduce the inference time of active approaches by taking as input, results from a passive inference. We also improve the stealthiness of the inference by reducing the number of invalid queries sent to the targeted implementation by means of a *divide-and-conquer* solution.

This Chapter is organized as follows: we first give some insights on our approach in Section 11.1 and describe in Section 11.2 the created state machine we use to illustrate the different steps of our approach. We then explain in Section 11.3 how we rely on our vocabulary inference work to obtain the vocabulary of each sub-parts of the protocol. Section 11.4 describes our solution to execute in parallel our inference process by means of Representatives Sequence of Symbols (RSS). Finally, Sections 11.5 and 11.6 successively details the inference of each protocol sub-part and our merging algorithm.

11.1 Big Picture

Our solution relies on the observation that a protocol exposes various protocol actions to its user. All these protocol actions participate to the general purpose of the protocol such as the authentication of the client or the creation of a directory in the FTP protocol. An action can be seen as a functional component of the protocol and denotes a subset of the protocol vocabulary and grammar. Indeed, our notion of *action frames* we relied on to infer the protocol vocabulary in Section 6.3.1 represents a valid path in the grammar of an action. We exploit this functional composition of protocols to divide the inference process in small blocks, each inferring the state machine of a protocol action. Obviously, our approach assumes that the targeted protocol grammar is the result of such composition.

More formally, the action $\mathcal{A}^{\mathrm{i}}$ represented by its state machine $\mathcal{M}^i = \langle \Sigma'_{Ii}, \Sigma'_{Oi}, Q_i, \lambda_i, \delta_i \rangle$ is an action of the protocol modeled by $\mathcal{M} = \langle \Sigma'_I, \Sigma'_O, Q, \lambda, \delta \rangle$ if and only if, its input and output alphabets are subset of the protocol alphabets ($\Sigma'_{Ii} \subseteq \Sigma'_I$, $\Sigma'_{Oi} \subseteq \Sigma'_O$), its states also participate in $\mathcal{M}$ ($Q_2 \subseteq Q$) and all its transitions exist in $\mathcal{M}$:

$$\forall s_i \in \Sigma'_{Ii}, \forall q_i \in Q_i, \begin{cases} \lambda_i(q_i, s_i) = \lambda(q_i, s_i) \\ \delta_i(q_i, s_i) = \delta(q_i, s_i) \end{cases} \tag{11.1}$$

Intuitively, if $\mathcal{M}^i$ is a subset of $\mathcal{M}$, its language denoted $\mathcal{L}_{M^i} = \{w \in \{\Sigma'_{Ii} \times \Sigma'_{Oi}\}^*\}$ is a subset of the language $\mathcal{L}_{\mathcal{M}}$ and is called a sub-language. By inferring the action sub-grammar we therefore infer a portion of the entire protocol grammar. By repeating this operation on all the protocol actions, we propose to combine them to obtain the protocol grammar.

To infer the grammar of each action, we first rely on our work described in Part I to infer its symbolic vocabulary. As a remainder, our vocabulary inference solution leverages semantic information to identify *action frames*. We rely on these action frames to identify input and output symbols that are related to the targeted protocol action. These symbols become the vocabulary of the action. Once we have the vocabulary of each action, we infer in parallel the state machine of each action using an active grammatical inference algorithm. Finally, we merge the inferred grammars to retrieve the state machine of the entire protocol.

Among required input, the user must provide a resetable implementation of the protocol and the value of few model parameters such as the w parameter described in section 10.2. In addition to these common requirements, our vocabulary inference step also need samples of communication traces annotated with the performed actions on the implementation that triggered their exchanges. We use these annotations to identify symbols that are related to the same action. As detailed in Section 6.2, these annotations can either be manually retrieved by the expert during the capture process or automatically through the instrumentation of the graphical interfaces or the OS, *i.e.* mouse, keyboard, button clicks, *etc.* For example, we successfully used the *android-hooker* [26] tool to automatically stimulate an android application and record all the graphical actions performed on its interface while we collected some network traces of its protocols.

11.2 Example Protocol to Illustrate our Approach

For sake of comprehension, we illustrate the application of our approach to infer the grammar of an example protocol given a set of traces that are supposed to be previously captured. We designed the state machine of this protocol (Figure 11.1) with four different actions, each illustrating a different aspect of our inference algorithm.

The first action of our protocol is the login action ($\mathcal{A}^{\texttt{LOGIN}}$). This action, executed by the user to authenticate, denotes a two step authentication schema, *i.e.* the user must provide a valid password after a valid username. If the user fails to provide a valid password after three attempts, its connection gets reseted to the initial state of the protocol. To illustrate how our solution can infer

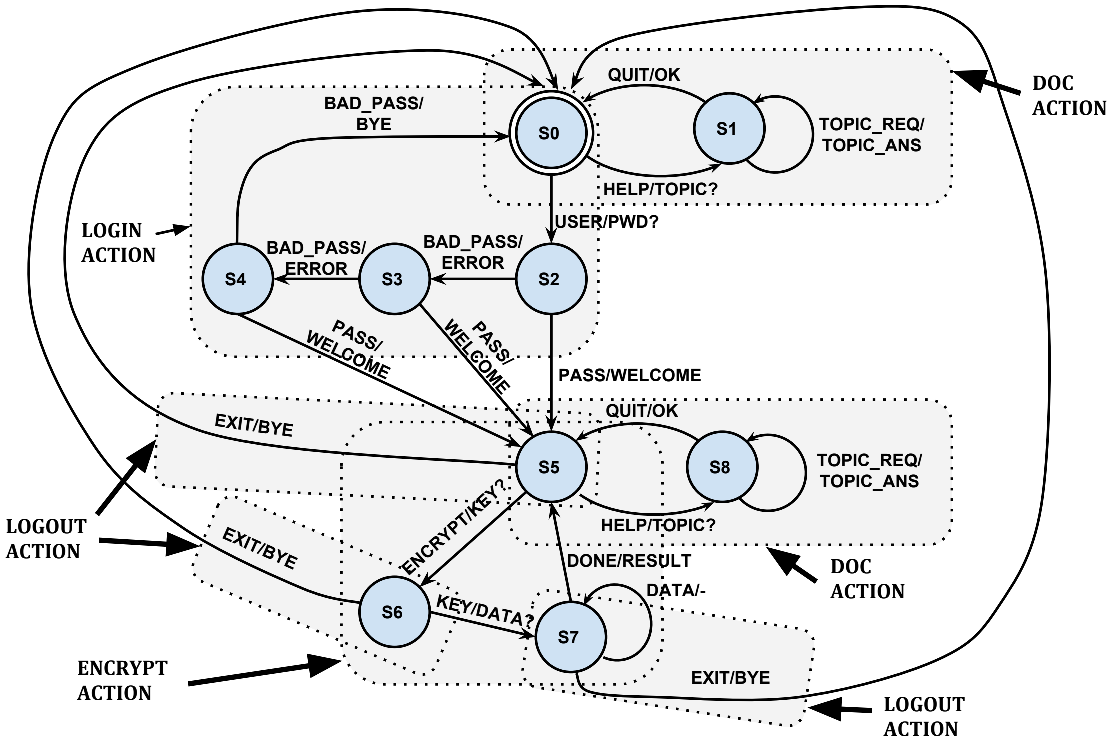


Figure 11.1 – State machine of the fake protocol we use to illustrate the steps of our inference approach.

transitions that are not present in captured traces, we deliberately do not include examples of this reset in the traces.

Our example protocol also offers a simple key based encryption action ($\mathcal{A}^{\texttt{ENCRYPT}}$). The user can provide a key (k) and a string of its choice ($clear$) and is returned with $k \oplus clear$. As illustrated on figure 11.1, this action is only available to authenticated users. We use this action to illustrate how our solution behaves on actions that are only available after the execution of others.

In addition, this protocol includes a documentation action ($\mathcal{A}^{\texttt{DOC}}$) allowing the user to obtain some help on the other actions of the protocol. This action can be assimilated to the "man page" of the protocol and is available both to authenticated and unauthenticated users. However, the traces we use in this example do not include any occurrence of the execution of this action by an authenticated user. Thus, we illustrate how our solution can infer that an action can be available from different states of a protocol even if the provided traces do not exhibit such behavior.

Finally, our example protocol includes another interesting feature: the authenticated user can interrupt the execution of the encrypt action to execute the logout action ($\mathcal{A}^{\texttt{LOGOUT}}$). Such interruption of an action by another one is complex to infer. It requires to consider that actions may have multiple output states, *i.e.* a state that accepts a symbol of a different action. To infer these output states, we include some parts of other action vocabularies in the inference process of each action. We give a more precise description of these parts latter in Section 11.4.

In the remainder of this section, we first detail how we retrieve the vocabulary of each action and then describe the three main steps of our hybrid inference algorithm: 1) the computation of the

Representative Sequence of Symbols (RSS) of each action, 2) the inference of each action state machine and 3) the merging algorithm we use to retrieve the final grammar.

11.3 Computing the Vocabulary of each Action

As explained previously, our approach leverages annotated symbolic traces to identify the vocabulary of the different actions accepted by the protocol. By symbolic trace, we hereby refer to a sequence of symbols denoting an observed protocol session. We follow our approach detailed in Section 6.2 to obtain such symbolic trace. In order to associate each symbol to one or more actions, the provided traces must be annotated. These annotations take the form of a set of chronological labels made of an action name and a timestamp denoting when each action was executed. Based on this timestamp, each label can be used to identify the first sent or received symbol after the execution of the action. As detailed in this section, we use these labels to cluster symbols according to their participation to one or more specific protocol action.

In our example, we use three different symbolic traces representing protocol exchanges generated by three different stimulation of the implementation. As illustrated by figure 11.2, the first trace denotes the sequential execution of the documentation action, the login action, the encrypt action and the logout action. The second trace is shorter and represents a user executing the documentation action and afterward failing to authenticate. Finally, the last trace denotes the execution of the login action followed by the logout action. Each trace is labeled with action names indicating an action starting points. For example, in the second trace illustrated in figure 11.2, the first four exchanges of input and output symbols were captured after the execution of the documentation action on the implementation.

We use these traces to compute the input and output vocabulary of each action. To achieve this, we analyze each annotated traces and consider that sent symbols (respectively received symbols) between the starting point of action $\mathcal{A}^{\mathtt{i}}$ and the starting point of the next action $\mathcal{A}^{\mathtt{i+1}}$ in the trace, belong to the input (resp. output) vocabulary of action $\mathcal{A}^{\mathtt{i}}$. It should be noted that a symbol can belong to the vocabulary of different actions. We successively apply this method on all the traces to retrieve the vocabulary of each action. For example, based on traces illustrated in figure 11.2, we compute the following input and output vocabularies of the documentation action ($\mathcal{A}^{\mathtt{DOC}}$):

$$\Sigma'_{I_{\mathtt{DOC}}} = \{\mathtt{HELP}, \mathtt{TOPIC\_REQ}, \mathtt{QUIT}\}$$

$$\Sigma'_{O_{\mathtt{DOC}}} = \{\mathtt{TOPIC?}, \mathtt{TOPIC\_ANS}, \mathtt{OK}\}$$

We leverage these action vocabularies in the inference process of the state machine of each action. Specifically, we run multiple instances in parallel of an active inference algorithm (or sequentially if a single non-threadable implementation is available), each instance being configured with the vocabulary of a given action. This way, each instance infers a portion of the protocol grammar. However, in some protocols, the execution of an action may require the prior execution of one or more other actions. Typically in our example protocol, the encryption action is only

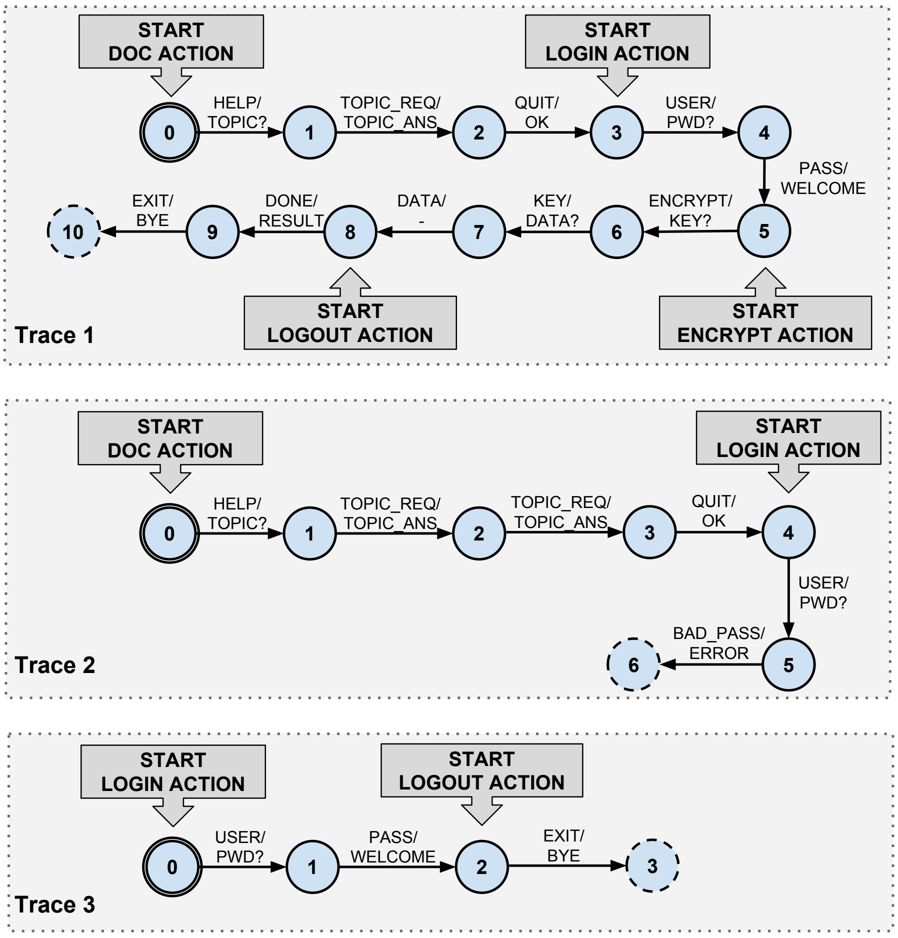


Figure 11.2 – The three annotated traces we use to infer the example protocol.

available if the client has previously executed a specific traversal path in the authentication action. We therefore ensure that each instance of the algorithm can traverse the other actions while inferring its action state machine. This way, besides the inference of its action state machine, each instance also learns how its action is interconnected with the other actions. We detail in the following section how we passively extract these traversal paths, we call Representatives Sequence of Symbols (RSS).

11.4 Inferring the Representatives Sequences of Symbols

The objective of this step is to infer the Representatives Sequence of Symbols (RSS) of each action. The RSS of an action denotes the shortest most observed traversal path in the grammar of the action. Such sequence starts with the first symbol participating in the action and ends with the last symbol of the action before another action is executed. We denote RSS: $\mathcal{A} \rightarrow (\Sigma'_I \Sigma'_O)^*$ a function computing the RSS of an action. $\text{RSS}_I : \mathcal{A} \rightarrow {\Sigma'_I}^*$ (respectively $\text{RSS}_O : \mathcal{A} \rightarrow {\Sigma'_O}^*$) returns the sequence of inputs symbols (respectively output symbols) participating in the RSS of an action.

An RSS denotes the execution of an action. Our merging algorithm uses such RSSes to identify equivalent states across different action state machines. If two states accept the same RSSes, we assume they are equivalent and merge them. To infer RSSes accepted by each state, we include them in the vocabulary of each action.

To compute the RSS of an action $\mathcal{A}^{\mathrm{i}} \in \mathcal{A}$, denoted RSS($\mathcal{A}^{\mathrm{i}}$), we passively infer the extended Prefix Tree Acceptor (ePTA) of each action. As defined by C. Higuera [63] a PTA is a tree-like DFA that only accepts the strings in the provided traces and in which common prefixes are merged together resulting in a tree-shaped automaton. To retrieve the most observed traversal path, we extend this definition of a PTA to create an ePTA by introducing a local occurrence probability on each transition. Thus, transitions are marked with $p(t/q)$, the probability the transition t occurs when current state is q. This probability occurrence is local which means $\forall q \in Q$, the set of states of the ePTA, $\sum_{t \in \phi(q)} p(t/q) = 1$ with $\phi : Q \rightarrow T^*$ a function returning the available transitions starting on a state.

To compute the ePTA of an action, we create an initial state and maintain a current state pointer initialized on it. We then use every provided traces to update it. For each trace, we sequentially play all its input and output symbols in the ePTA. If the current symbol belongs to the vocabulary of the action, we create, if it does not exist, a transition starting on the current state pointer. This transition is labeled with the current symbol and ends on a new state that becomes the new current state pointer. If a transition labeled with the same symbol and starting on the current state pointer already exists, we update the current state pointer on its ending state. In both case, we update the probability occurrence of the transition. This operation is repeated while the current symbol belongs to the vocabulary action. If not, the current state pointer is reseted to the initial state of the ePTA and we continue our algorithm on remaining symbols. Listing 11.1 denotes the algorithm we use to build the ePTA of an action.

```
func buildEPTA(Trace[] traces, Action a):
   Node initialN = Node()
   foreach (Trace trace in traces):
       Node currentN = initialN
       foreach (Symbol symbol in trace.symbols):
           if (symbol in a.vocabulary):
               currentN = addSymbol(currentN,symbol)
           else:
               currentN = initialN
   return initialN
func addSymbol(Node node, Symbol s):
   foreach (Transition trans in node.transitions):
       if (trans.symbol == s):
           transition.occurrence += 1
           return transition.nextNode
   Node node = Node()
   Transition newTrans = Transition(node, s, node)
   return node
```

Listing 11.1– Algorithm used to build the ePTA of an action.

For example, this algorithm computes the ePTA of the documentation action ($\mathcal{A}^{\text{DOC}}$) as illustrated in figure 11.3. The first trace is used to initiate the ePTA, only symbols related to the documentation action are retained. We then update the ePTA by successively applying the second and the last trace. The second trace starts with the same sequence of input and output symbols. However, its fifth symbol is not the `QUIT` symbol as in the current ePTA but a `TOPIC_REQ` symbol. Thus, a new branch is created and added on the fifth state of the ePTA. The third trace does not contain symbols related to the documentation action and thus its application does not update the ePTA of the documentation action. Contrary to IO Automaton and especially our Symbolic Mealy Machine, each transition of an ePTA denotes either an input or an output symbol. In the figure, we make a distinction between input and output transitions using dashed and plain arrows. We also annotate each transition with their occurrence frequency.

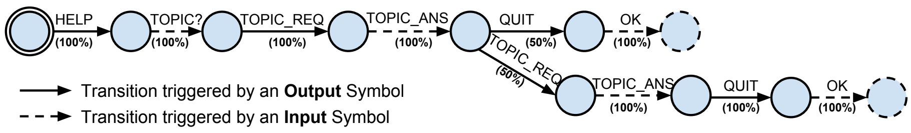


Figure 11.3 – ePTA of the doc action ($\mathcal{A}^{DOC}$).

We then compute the RSS of an action by identifying the shortest most frequent path in its ePTA. To achieve this, we compute the frequency of each path and retain the most frequent. If multiple paths have the same frequency, we keep the shortest one. This operation is repeated for every actions of the protocol. For example, based on the computed ePTA of the documentation action illustrated in Figure 11.3, it is straightforward to compute:

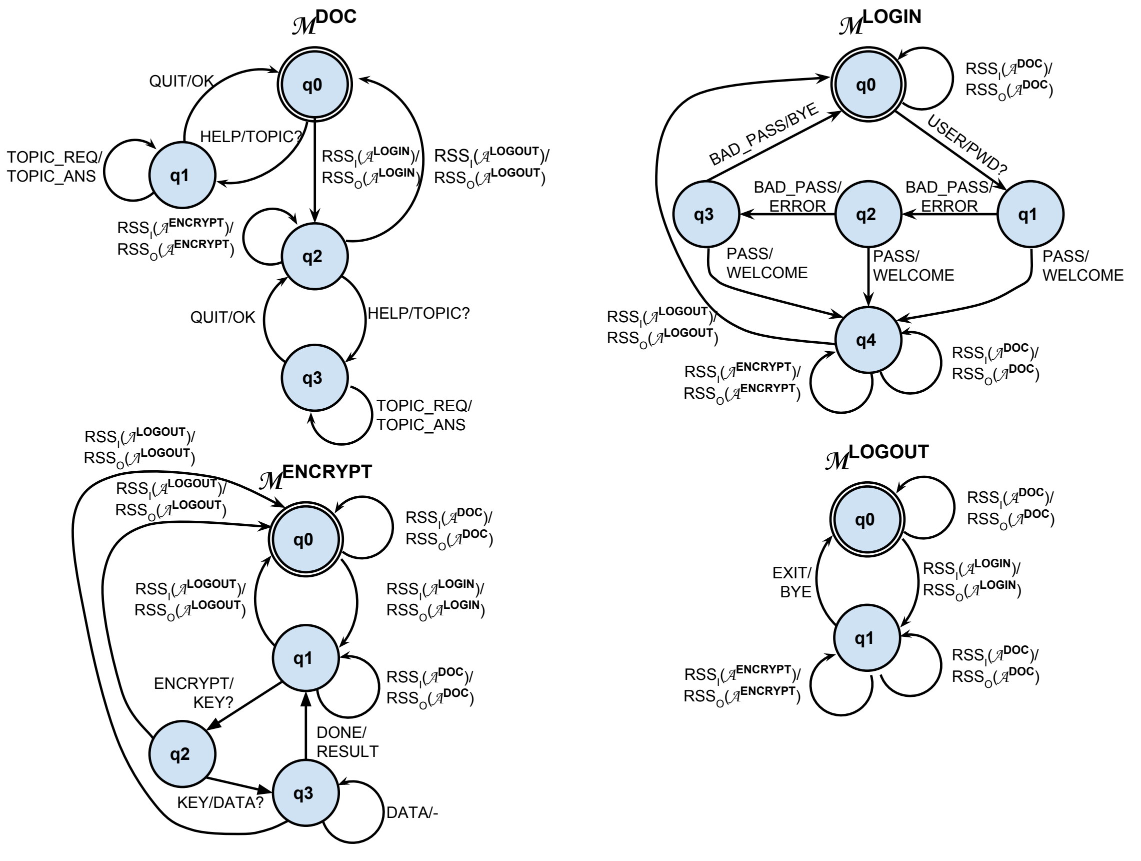
$$\text{RSS}_I(\mathcal{A}^{\text{DOC}}) = [\texttt{HELP}, \texttt{TOPIC\_REQ}, \texttt{QUIT}]$$

$$\text{RSS}_O(\mathcal{A}^{\text{DOC}}) = [\texttt{TOPIC?}, \texttt{TOPIC\_ANS}, \texttt{OK}]$$

11.5 Inferring Action State Machines

The objective of this step is to infer the state machine of each action of the the protocol. To achieve this, we execute a dedicated L^* inference instance for each action. Thereby, each instance objective is to infer a portion of the whole protocol state machine. To limit the inference scope to the action state machine, each instance is configured with a specific input vocabulary mostly composed of the action input vocabulary. For example, to infer the state machine of the documentation action, we include the following symbols in the vocabulary of its L^* instance: `HELP`, `TOPIC_REQ` and `QUIT`.

In addition to these symbols, we also add the RSS_I symbols of each other action. This way, the inferred action state machine denotes how it is interconnected with other action state machines. It also permits to infer the interruption of the inferred action state machine by other actions. We



$\mathcal{M}$DOC
q0
q1
q2
q3
QUIT/OK
HELP/TOPIC?
TOPIC_REQ/ TOPIC_ANS
RSS_I($\mathcal{A}$LOGIN)/ RSS_O($\mathcal{A}$LOGIN)
RSS_I($\mathcal{A}$LOGOUT)/ RSS_O($\mathcal{A}$LOGOUT)
RSS_I($\mathcal{A}$ENCRYPT)/ RSS_O($\mathcal{A}$ENCRYPT)
QUIT/OK
HELP/TOPIC?
TOPIC_REQ/ TOPIC_ANS
$\mathcal{M}$LOGIN
RSS_I($\mathcal{A}$DOC)/ RSS_O($\mathcal{A}$DOC)
BAD_PASS/BYE
USER/PWD?
BAD_PASS/ ERROR
BAD_PASS/ ERROR
PASS/ WELCOME
PASS/ WELCOME
PASS/ WELCOME
RSS_I($\mathcal{A}$LOGOUT)/ RSS_O($\mathcal{A}$LOGOUT)
RSS_I($\mathcal{A}$ENCRYPT)/ RSS_O($\mathcal{A}$ENCRYPT)
RSS_I($\mathcal{A}$DOC)/ RSS_O($\mathcal{A}$DOC)
$\mathcal{M}$ENCRYPT
RSS_I($\mathcal{A}$LOGOUT)/ RSS_O($\mathcal{A}$LOGOUT)
RSS_I($\mathcal{A}$LOGOUT)/ RSS_O($\mathcal{A}$LOGOUT)
RSS_I($\mathcal{A}$DOC)/ RSS_O($\mathcal{A}$DOC)
RSS_I($\mathcal{A}$LOGOUT)/ RSS_O($\mathcal{A}$LOGOUT)
RSS_I($\mathcal{A}$LOGIN)/ RSS_O($\mathcal{A}$LOGIN)
RSS_I($\mathcal{A}$DOC)/ RSS_O($\mathcal{A}$DOC)
ENCRYPT/ KEY?
DONE/ RESULT
KEY/DATA?
DATA/-
$\mathcal{M}$LOGOUT
RSS_I($\mathcal{A}$DOC)/ RSS_O($\mathcal{A}$DOC)
EXIT/ BYE
RSS_I($\mathcal{A}$LOGIN)/ RSS_O($\mathcal{A}$LOGIN)
RSS_I($\mathcal{A}$ENCRYPT)/ RSS_O($\mathcal{A}$ENCRYPT)
RSS_I($\mathcal{A}$DOC)/ RSS_O($\mathcal{A}$DOC)

Figure 11.4 – Inferred action state machines of the documentation action ($\mathcal{M}^{doc}$), the login action ($\mathcal{M}^{login}$), the encrypt action ($\mathcal{M}^{encrypt}$) and the logout action ($\mathcal{M}^{logout}$).

therefore use the following input vocabulary to infer the state machine of the documentation action:

$$\mathcal{L}^*_{\mathtt{DOC}} \cdot \Sigma'_I = \{\mathtt{HELP}, \mathtt{TOPIC\_REQ}, \mathtt{QUIT}, \mathrm{RSS}_I(\mathcal{A}^{\mathtt{login}}), \\ \mathrm{RSS}_I(\mathcal{A}^{\mathtt{ENCRYPT}}), \mathrm{RSS}_I(\mathcal{A}^{\mathtt{LOGOUT}}\}$$

Applied to our example, the inference of the documentation state machine produces $\mathcal{M}^{\mathtt{DOC}}$ illustrated in Figure 11.4. The inferred state machine is made of four states, labeled *q0*, *q1*, *q2* and *q3*. The first two denotes the execution of the documentation action by an unauthenticated user while the others, its execution by the authenticated user. The two possible executions of the documentation action are similar, *i.e.* the user sends the `HELP` symbol and then requests for a specific help subject by emitting `TOPIC_REQ` symbols. Finally, the user can stops the documentation action by emitting the `QUIT` symbol. The inference process identified the possibility for an authenticated user to execute the documentation action, even if the provided traces did not mention it. The inferred state machine also shows that the documentation action denotes a single output state.

Besides the complete inference of the transitions participating in the encrypt action, also illustrated in Figure 11.4, our approach has successfully inferred a transition on state $Q2$ that is triggered by the symbol $\mathrm{RSS}_I(\mathcal{A}^{\mathtt{LOGOUT}})$ and that produces the symbol $\mathrm{RSS}_O(\mathcal{A}^{\mathtt{LOGOUT}})$. Our merging algorithm relies on such transition to merge action state machines. Finally, the inferred state machine of the login action ($\mathcal{M}^{\texttt{LOGIN}}$) successfully denotes the two steps authentication including the limitation over the number of successive `BAD_PASS` symbols that can be emitted.

The inference of an action state machine requires no information from any of the other inference instances. Thus, every L^* instances can be executed in parallel to reduce the total computation time. To do so, the user must either have access to multiple implementations of the protocol or to a single one that can handle, independently, multiple client connections. Each instance of the L^* algorithm has access to a common query cache. Before the execution of the first instance of L^* , this cache is pre-filled using traces which were previously captured and used during the passive vocabulary inference step. This cache is then updated by each instance during the active inference phase.

In addition to its primary usage, we also use the cache to compute the reaction time of each transition. The way L^* works ensures that every transitions of the inferred state machine is the result of at least one query stored in the cache. When the inference of a state machine is completed, we compute the reaction time of its transitions by analyzing the cache. This cache stores all the observed times between the emission and the reception of a symbol corresponding to each inferred transition. Thus, we use it to compute the mean and the standard deviation of the reaction time attached to each transition.

Inferred action state machines describe the complete internal structure of their actions but also denote their interconnections with the state machines of other actions. The *merging step* leverages these interconnections to identify and merge equivalent states across different action state machines.

11.6 Merging Sub-Grammars

Finally, we obtain the grammar of the protocol by merging the action state machines inferred in the previous step. To achieve this, we randomly select one of the action state machine, we call the *target state machine* and recursively merge it with others action state machines. By merging, we refer to the creation of new states and transitions so that the resulting automaton accepts all valid message sequences of the protocol. This operation relies on the identification or the creation of an equivalent transition in the target state machine for each transition of the action state machine that is being merged. Section 11.6.1 details our solution to identify equivalent transitions and states across two state machines. Section 11.6.2 describes our algorithm that leverage our definition of transition equivalency to recursively extend the target state machine with the different action state machines.

11.6.1 Transition and State Equivalencies

A transition in the action state machine and a transition in the target state machine are equivalent if their starting and ending states are equivalent and if both their input and output symbols are the same. By definition, two states are equivalent if they produce the same output strings for any input strings. Another definition of state equivalency comes from the definition of DFA, it states that given two input symbols $a \in \Sigma'_I$, $a' \in \Sigma'_I$ and two transitions $\textcircled{\scriptsize q0} \xrightarrow{\texttt{a/b}} \textcircled{\scriptsize q1}$, $\textcircled{\scriptsize q0'} \xrightarrow{\texttt{a'/b'}} \textcircled{\scriptsize q1'}$, $(a \equiv a', q0 \equiv q0') \Rightarrow q1 \equiv q1'$. Our merging algorithm relies on this definition to identify most equivalent states in its merging process. However, this definition cannot be use to identify a state equivalency between two states $q1$ and $q1'$ of different state machines if the transition $\textcircled{q0'} \xrightarrow{\texttt{a'/b'}} \textcircled{q1'}$ doesn't exist yet and needs to be created. To address this case, we propose a heuristic that leverages the semantic definition of each action state machine to identify that two states are equivalent.

In our additional state equivalency definition, we refer to as a state equivalency by context, we assume that two states are equivalent if they accept the same sequences of RSSes. In our model, the transitions triggered by an RSS_I denotes the execution of an action. Furthermore, if this transition generates the associated RSS_O, it represents the valid execution of the action. Thus, the context of a state is the set of valid RSS exchanges it accepts. We model the context of a state with an equivalent regular expression. For example, the state $q0$ of the documentation action ($\mathcal{A}^{\text{DOC}}$) and the state $q0$ of the login action ($\mathcal{A}^{\text{LOGIN}}$) illustrated in Figure 11.4 have the same context. This context represents the sequences of actions that an unauthenticated user can trigger. Thus, we model it by means of a regular expression denoting that an unauthenticated user can execute the documentation action and the login action which gives access to the encrypt action, the documentation action and the logout action:

$$((\mathcal{A}^{\text{DOC}})^*(\mathcal{A}^{\text{LOGIN}})((\mathcal{A}^{\text{DOC}})^{\{,1\}}(\mathcal{A}^{\text{ENCRYPT}})^{\{,1\}})^*(\mathcal{A}^{\text{LOGOUT}}))^*$$

On the other hand, these states have a different context than the context of state *q1* of the logout action ($\mathcal{A}^{\text{LOGOUT}}$) which can be modeled with the following regular expression:

$$(((\mathcal{A}^{\text{DOC}})^{\{,1\}}(\mathcal{A}^{\text{ENCRYPT}})^{\{,1\}})^*(\mathcal{A}^{\text{LOGOUT}})(\mathcal{A}^{\text{DOC}})^*(\mathcal{A}^{\text{LOGIN}})^{\{,1\}})^*$$

In practical, to check if two states are equivalent by context, we therefore compare the sequences of RSS symbols they accept under a predefined *horizon*. The horizon of a context denotes the maximum length of each sequence of RSS symbols included in the context. For example in Figure 11.4, the context of state *q0* in the documentation state machine accepts five different sequences of RSS symbols under an horizon of two:

$$\begin{array}{l}
\{ \\
\ [\text{RSS}(\mathcal{A}^{\text{DOC}}), \text{RSS}(\mathcal{A}^{\text{DOC}})]; [\text{RSS}(\mathcal{A}^{\text{DOC}}), \text{RSS}(\mathcal{A}^{\text{LOGIN}})]; \\
\ [\text{RSS}(\mathcal{A}^{\text{LOGIN}}), \text{RSS}(\mathcal{A}^{\text{DOC}})]; [\text{RSS}(\mathcal{A}^{\text{LOGIN}}), \text{RSS}(\mathcal{A}^{\text{ENCRYPT}})]; \\
\ [\text{RSS}(\mathcal{A}^{\text{LOGIN}}), \text{RSS}(\mathcal{A}^{\text{LOGOUT}})] \\
\}
\end{array}$$

In our experiments presented in Chapter 12, we use an horizon of five which is enough to compute detailed enough contexts. However, we detail in the following that this value must be increased if inconsistent equivalencies are found while merging action state machines.

11.6.2 Merging the Target State Machine with an Action State Machine

As explained previously, our goal is to extend the target state machine in a way that it accepts all the sequences of symbols accepted by action state machines. To achieve this, we successively pick and merge each action state machine with the target state machine.

Figure 11.5 gives an example of two state machines we want to merge. It shows $\mathcal{M}^{TARGET}$ a target state machine and $\mathcal{M}^{ACTION}$ an action state machine. $\mathcal{M}^{TARGET}$ contains two transitions triggered by input symbols *A* and *B*. These transitions generate two different output symbols named *1* and *2*. $\mathcal{M}^{ACTION}$ contains three transitions triggered by input symbols *B*, *C* and *D* and generates three different output symbols named *2*, *3* and *4*. Both state machines denotes an equivalent context named *C1* on states *q2'* and *q2*. In the following, we illustrate our merging algorithm with this example.

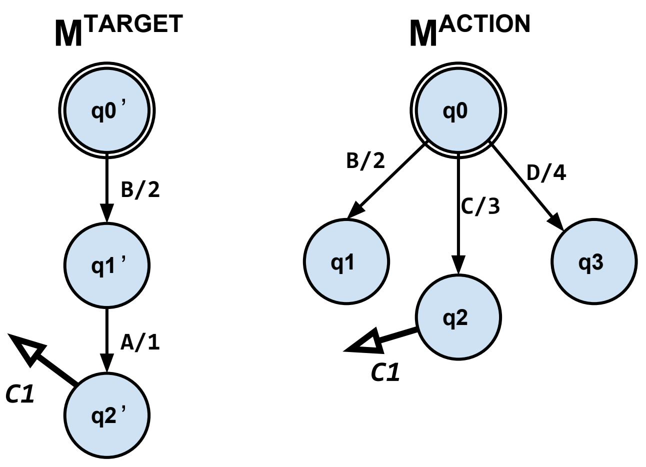


Figure 11.5 – Simple example that illustrates how we merge an action state machine with a target state machine.

Our merging algorithm relies on a depth-first search algorithm to traverse all the transitions of the action state machine that is to be merged. For each transition we traverse, we create an equivalent one in the target state machine if it does not exist.

By construction, initial states of every action state machines are equivalent. It comes from the fact that a DFA possesses a single initial state and that we used the same reset operation when we inferred each action state machine. Coupled with the use of a depth-first search, it ensures that we already identified an equivalent starting state in the target state machine for every transition we traverse. Thus, we compare all the transitions accepted by the equivalent starting state in the target state machine with the transition we are traversing. If we traverse the transition $\textcircled{qi} \xrightarrow{\texttt{a/b}} \textcircled{qj}$ in the action state machine that we merge and if it exists a transition $\textcircled{qi'} \xrightarrow{\texttt{a'/b'}} \textcircled{qj'}$ in the target state machine with $qi \equiv qi'$, we memorize that $qj \equiv qj'$ in regards to the first definition of state equivalency we detailed in Section 11.6.1. If no such transition can be found in the target state machine, we apply our state equivalency by context to identify an equivalent state. Finally, if no equivalent state can be found in the target state machine, we create it.

The first case is illustrated in our example when we traverse $\textcircled{q0} \xrightarrow{\texttt{B/2}} \textcircled{q1}$ and check for an equivalent transition in the target state machine. To achieve this, we first identify an equivalent state for *q0*. By definition, initial states are equivalent so we can easily spot that states *q0'* $\equiv$ *q0*. We then search for a transition accepted by *q0'* that has the same symbols, *i.e.* `B`. In our example, such transition exists: $\textcircled{q0'} \xrightarrow{\texttt{B/2}} \textcircled{q1'}$. We therefore memorize that $q1' \equiv q1$ and continue our depth-first search algorithm.

If no equivalent transition is found, we create it. To prevent from duplicating states, we first search for an equivalent ending state in the target state machine. We use our definition of state equivalency by context stated in 11.6.1 to identify it. To be equivalent, a state of the target state machine and the ending state of the transition must share the same non-empty context. We also check that the target state has not been previously memorized as being equivalent to another state in the action state machine. If both conditions are fulfilled, the target state becomes the ending state of a new transition.

This situation is illustrated in our example when we traverse $\textcircled{q0} \xrightarrow{\texttt{C/3}} \textcircled{q2}$. No equivalent transition can be found in the target state machine, *i.e.* no transition triggered by symbol *C* is accepted by *q0'*. We therefore try to apply our definition of state equivalency by context to identify an equivalent state to $q2$ in the target state machine. Since *q2* denotes a non-empty context (*C1*), we search for a state that shares the same context in the target state machine. This search returns *q2'*. We conclude that *q2* $\equiv$ *q2'* and create the transition $\textcircled{q0'} \xrightarrow{\texttt{C/3}} \textcircled{q2'}$ in the target state machine.

Following our definition of state equivalency by context, multiple states in the target state machine can be identified as equivalent to a single state in the action state machine. It happens when multiple states of the target state machine that we did not yet traversed share the same context. When this case is encountered, we first recompute the context of these states with an extended horizon. If despite such effort, states are still equivalent we rely on another solution: we do not create the transition now and continue the merging process. When all the remaining and accessible transitions of the state machine are traversed, we merge a second time the action state machine. However this time, we rely on state equivalencies memorized during the first attempt. This way, when we try to recreate the transition, we can filter out states that we identified as equivalent to other states in the action state machine. If in the worst case scenario, our solution does not resolve the issue, we create a new state in the target state machine. This way, we ensure the correctness of the inferred state machine in the expense of its completeness.

Finally, if our two definition of state equivalency returned no equivalent state, we create a new state in the target state machine a,d memorize it as equivalent to the transition ending state. We then create a transition between the equivalent initial state and this newly created state. This transition is labeled with the same symbols than the transition we are traversing.

This situation is illustrated in our example when we traverse $\textcircled{q0} \xrightarrow{\texttt{D/4}} \textcircled{q3}$. No equivalent transition can be found in the target state machine and *q3* has no context which prevent the identification of an equivalent ending state in the target state machine. We therefore create a new state (*q3'*) and an equivalent transition in the target state machine, $\textcircled{q0'} \xrightarrow{\texttt{D/4}} \textcircled{q3'}$. Finally, the result of the merging process of $\mathcal{M}^{TARGET}$ and $\mathcal{M}^{ACTION}$ is illustrated on Figure 11.6.

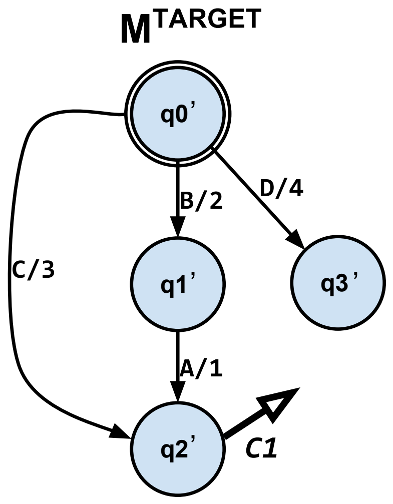


Figure 11.6 – Target state machine obtained after merging the two state machines illustrated on Figure 11.5.

We explained in this Chapter the solution we propose to learn the grammar of a protocol by means of an FSM decomposition. As described, our approach relies on our assumption that protocols state machines can be described under a compositional structure of smaller sub-grammars, one for each action of the protocol. This decomposition allows us to execute in parallel the inference process while combining an active and a passive approach to limit the computation time and to increase the inference stealthiness. In the following, we describe our evaluation of our solution.

Chapter 12

Evaluation

To conduct this evaluation, we compared our results against those computed by the traditional version of L^* exposed by the LearnLib [130] framework. We performed three different experiments, each applied on a different protocol: IRC, SAMBA and the C&C protocol of a botnet. Among the retained protocols, two are famous known protocols while the last one is an undocumented protocol used by a recent version of the pbot botnet. This comparison shows that our approach is effective to compute a good approximation of the targeted protocol grammar while being faster and stealthier than previous work.

In the remainder, Section 12.1 gives some insights over the selected protocols and the datasets we used. We then detail in Section 12.2 the metrics we used and in Section 12.3 the implementations we developed to perform the evaluation. We conclude in Section 12.4 with a discussion on obtained results.

12.1 Datasets

Our comparative study relies on three protocols. We selected these protocols to evaluate our approach on protocols of various sizes in terms of symbols and states. The characteristics of these protocols are provided in Table 12.1. This table shows the number of states in each protocol identified by the L^* algorithm. Another characteristic is the number of symbols declared in the specification of each reversed protocol vocabulary. Finally, it also shows the number of actions we considered when reversing them. We provide more information on these actions latter in this Section.

The first protocol is the IRC protocol which vocabulary is made of almost a hundred of symbols. The second protocol we used in our comparative study is the SAMBA v2 protocol. This protocols is used by different work to evaluate the quality of their algorithms [43, 41, 142]. Indeed, its complexity makes of it a good candidate to evaluate the efficiency of grammatical inference algorithms. We also evaluated our approach by reversing the protocol of a famous botnet used in a recent version of the PBot malware [137].

As explained in Section 11.1, our approach requires various inputs: an implementation of the protocol, its vocabulary and some annotated traces. To infer the IRC protocol, we installed and

Protocol	Number of states	Number of Symbols	Number of actions	Implementation
IRC	8	97	4	Miniircd 0.4
Samba v2	17	36	3	Linux Samba 4.0
Pbot	8	32	3	Malware found in the wild

Table 12.1 – Details on inferred protocols.

configured the latest version of the Miniircd IRC server [1]. We used the official Linux implementation of the SAMBA v2 server to infer its protocol. Finally, we searched for infected web servers on the Internet and downloaded from one of them, a version of the PBot malware. We performed a security audit of its code to identify and neutralize potential dangerous features and deployed the malware in a specifically confined environment.

We also explained in Section 11.1 the necessity to have access to the protocol vocabulary to infer its grammar. To reduce the impact of vocabulary errors in our evaluation, we performed a manual extraction of the IRC and SAMBA vocabularies out of their client implementations. We relied on the Java "IRC Martyr" [2] client library for the IRC protocol and used the PySMB [3] client for the SAMBA protocol. This way, we ensure the quality of the vocabulary which ease the interpretation of the results returned by our evaluation of our grammatical inference solution. However, we had no access to a client implementation to obtain the vocabulary of the botnet protocol. We therefore applied the solution we proposed in Part I to reverse its vocabulary. To ensure its correctness, we verified our results by means of a source code analysis of the malware.

Lastly we need sample annotated traces denoting common usages of these protocols. To retrieve them, we first identify the different actions of each protocol. We rely on their implementations to identify them, a technique used in [24]. We then follow the approach detailed in Section 11.3 and manually stimulate the client implementation of each targeted protocol while capturing the generated network exchanges.

For the IRC protocol, we identified four different kind of actions accepted by the client implementation: connecting to a server, editing user information, joining an IRC channel and disconnecting from a server. We therefore captured the traffic while executing these actions in different ways which brought us three traces. The first trace is the result of a bad authentication process on the server, *i.e.* we voluntary gave a bad server password. The second trace contains the symbols exchanged while connecting to the IRC server, sending a message to a channel and then exiting the server. The last IRC trace we used contains the symbol exchanged while connecting to the IRC server, executing various user configuration commands such as changing our user status and then joining a channel. On this channel, we modified the channel topic and sent some channel messages. We then exited the channel and the server.

For the SAMBAv2 protocol, we identified three different kind of actions. The first concerns the connection to the SAMBA server, the second covers file accesses while the third concerns

1. Miniircd is available at `https://github.com/jrosdahl/miniircd`
2. IRC Martyr is available at `http://martyr.sourceforge.net/`
3. PySMB is available at `https://miketeo.net/wp/index.php/projects/pysmb`

connection management. We used the same approach we followed for the IRC protocol and succesively executed these actions such as connecting to the SAMBA server, navigating in a directory, listing its content and querying some file details to obtain the trace.

Regarding the botnet protocol, we first identified the various features offered by the botnet such as the execution of system commands on an infected host or making it execute an UDP flood. We based our inference on three actions. The first action denotes all the symbols related to the connection to a botnet host. The second action, all the symbols related to the disconnection from the infected host while the third action regroups all the symbols related to the botnet commands. To obtain the annotated traces, we infected one of our host with this botnet and simulated the botnet master to send various orders to the infected host. The captured trace shows a connection to the infected host and the successive emission of all the botnet orders we identified before we logout.

12.2 Metrics

To measure and compare the effectiveness of our inference algorithm against state-of-the-art L^* algorithm, we need to define metrics. Specifically, we want to demonstrate that our approach can infer a good approximation of the final grammar of the protocol while being stealthier and faster than state of the art. Thus, we first evaluate the quality of our inferred state machine by means of two different methods: we check the correcteness of the inferred state machine and measure its completness. We then propose various metrics covering the inference time, the number of sent symbols and the number of erroneous queries required to infer a protocol.

We expect our inferred grammar to be correct while having a high completeness in comparison to the grammar inferred by the traditional L^* algorithm. We say our model is correct if its state machine only produces valid sequences of symbols. On the other hand, our inferred grammar is said complete if all the sequences of symbols accepted by the protocol grammar are also accepted by our grammar.

To check if the inferred grammar is correct, we use a random walk algorithm to produce a thousand of random paths that are accepted by the inferred grammar. We then compare the result of their submissions to an implementation of the protocol. Our inferred grammar is said correct if all the generated paths are also accepted by the protocol implementation. Each path is made of at most 50 input symbols obtained after randomly traversing the inferred state machine with a reset probability of 1%. This configuration ensures a high coverage of the protocols state machine.

To evaluate the completeness of our grammar, we compare all the transitions inferred by the traditional L^* algorithm against the transitions inferred using our approach. Our objective is to identify the transitions we missed and discuss the reasons for it. In particular, we determine the importance of each transition we missed. Indeed, we consider that the transitions that lead to different states in the protocol are the most important because they give access to a new context in the protocol. Conversely, self-loop transitions that are caused by invalid sequences of symbols appear to us as less important. Indeed, the implementation of a default policy for invalid sequence of symbols can represent them. For example, a default policy can specify that for each $a \in \Sigma'_I$ and for each $q \in \mathcal{Q}$ where $\theta(q, a) = \emptyset$, we create a self-loop transition $\textcircled{q} \xrightarrow{a/\epsilon} \textcircled{q}$.

As described in Section 1.2.2, one of the key aspect in defensive security is fast response time to new threats. However, our evaluation has revealed that current work relying on L^* can take hours to infer complex protocols such as SAMBA v2 protocol which required ten hours of active stimulation of its implementation. Therefore, we believe that reducing the grammatical inference time is an important goal. Since our approach participates in this objective, we compare the inference time required by the different approaches.

In addition to the inference time, we also want to measure the stealthiness of the compared approaches. To achieve this, we propose five metrics we detail in the following.

The first metric focuses on the number of symbols sent to the implementation. We believe the more symbol is sent by an inference algorithm the less stealthy it is. Thus, we measure the number of symbols received by an implementation i ($\mathcal{N}_S^i$). Besides, the number of sent symbols per seconds is also a factor that is often considered in protocol protections such as anti-flooding. To cover this aspect, we measure the average density of symbols sent to an implementation i per second ($\mathcal{D}_s^i$).

Another common detection technique used in anti-inference protections relies on the number of protocol errors made by a client of a protocol implementation. If this number reaches a given threshold, the implementation can trigger anti-inference techniques. For instance, we observed such protection in the implementation of the Ventrilo protocol [24] which bans users when they do too many protocol errors. To measure this, we first identify in each protocol the symbols that are sent by the implementation when it receives an invalid sequence of symbols. For the IRC and the SAMBAv2 protocol, we used their specifications to identify them. For the botnet protocol, we observed that when an infected host receives an invalid sequence of symbols it does no answer while he always does when the sequence is valid. Thus, if no answer follows the emission of a symbol, we assume that the symbol was erroneous. Based on this, we measure the number of erroneous symbol received ($\mathcal{N}_{es}^i$) and compute the average density of erroneous symbol per second sent by the implementation ($\mathcal{D}_{es}^i$).

Some protocol implementations also monitor the number of connections made by a client to its implementation. If too many connections are opened, the implementation detects its inference and can trigger protection techniques. For example, IRC servers such as UnrealIRCd[4] implements a "throttling" protection method that limits how fast a user can disconnect and then reconnect to it. In the case of active grammatical inference, a new connection is opened with the target implementation for each query. We therefore count the number of queries sent to each implementation ($\mathcal{N}_q^i$) to measure the number of connections. The fewer queries we observe, the stealthier the approach is.

Thus, the metrics we propose cover both the quality of the inferred state machine and the impact of its inference on the protocol implementation. The quality of the inferred state machine is evaluated by means of our correctness and completeness measures. To evaluate the impact of the inference process over the protocol implementation, we measure the number of queries, the number of symbols and the number of erroneous symbols that are triggered by each inference algorithm. We also measure the computation time of the different approaches and the density of symbols and

4. UnrealIRCd's homepage: `http://www.unrealircd.com`

erroneous symbols per second to estimate the inference stealthiness.

12.3 Implementations

In this Section, we present the two implementations we developed for the experimental phase.

To ensure a fair comparison between our approach and the traditional L^* algorithm, we implemented our Action-Based inference algorithm by extending the LearnLib framework [130]. LearnLib is an open-source library in Java that implements the L^* algorithm. This way, our approach relies on the same version of the L^* algorithm than the one we compared to. Besides, we executed all the experimentation on the same computer.

To identify the best parameters to configure the traditional L^* algorithm and our approach, we executed a set of inference process on the IRC protocol with different parameter values. We selected the IRC protocol because the inference of the other protocols with different parameters would have required many days of computations to complete. The objective of the first calibration step to experimentally identify the parameter values that returns the best results given our metrics. We considered the algorithm to find counter-examples and its parameters along with the algorithm used to handle the observation table. We tested all the different algorithms provided by the LearnLib implementation. We retained the Classic L^* implementation to handle the observation table conjointly with the RandomWalk Equivalence Algorithm. This last algorithm takes two different values in parameters, a restart probability and the maximum length of a walk in the hypothesis state machine. We experimentally identified that best results were obtained with a restart probability of 5% and a maximum length walk $L = |H_s||\Sigma_I^{'}| * 100$ with $|H_s|$ denoting the number of states in the current hypothesis state machine and $|\Sigma_I^{'}|$ the number of symbols in the input vocabulary.

Another important factor in such experimentation is the System Under Learning (SUL) driver that interconnects the inference algorithm with the targeted implementation. As described in Section 5.2, this code abstracts the received messages sent by the implementation into symbols that can be handled by the inference algorithm. On the other way, it specializes the symbols sent by the inference algorithm to the implementation into valid messages. The SUL driver is also in charge of opening the communication channel with the implementation, closing it after the execution of a query and resetting the implementation to its initial state. We expect that a message may be sent in portions. We therefore implemented a timeout mechanism in the SUL driver to handle fragmented messages. However, as explained in Section 3.2.2, this mandatory timeout highly impacts the inference computation time. To reduce this impact when the implementation answers rapidly while not missing late answers, we used two different timeout values, a short timeout (ST) and a long timeout (LT). Once the SUL driver has sent a message to the implementation, it waits for fast answers during a short period of time represented by ST. If no messages were received after ST, the SUL driver waits a longer period of time (LT) for late messages. Received messages are then abstracted into their respective symbols using our knowledge over the protocol vocabulary. If the received message cannot be abstracted, a specific `UNKNOWN_SYMBOL` symbol is created. In both cases, the symbol is returned to the inference algorithm. On the other hand, if no message has been received, the SUL driver returns to the inference algorithm a specific symbol denoted

`EMPTY_SYMBOL`. The different timeout values we used in our experimentation are detailed in table 12.2. Timeouts for the Botnet protocol are longer as some of its commands imply the execution of network requests that can take times to complete (*i.e.* sending emails, scanning a TCP port, *etc.*). On the contrary, the SUL drivers for the SAMBAv2 and the IRC protocol use shortest timeout values since none of the identified actions required a long computation time (*i.e.* listing directory contents, traversing folders, *etc.*).

Timeout per Protocol	IRC	SAMBA v2	PBot
Short timeout (ms)	100	50	500
Long timeout (ms)	200	100	1 000

Table 12.2 – Timeout values used for each protocol.

12.4 Experimental Results

In this section, we present the conclusions of our experimental comparative study of our approach against state-of-the-art inference algorithm. To achieve this, we rely on metrics we detailed in Section 12.2. First, we check the correctness and evaluate the completeness of our inferred state machines. We then compare the inference times and the inference stealth of our approach against traditional L^* inference process.

12.4.1 State Machines Correctness and Completeness

We expect inferred state machines to be both correct and as complete as possible. This means they accept no invalid transitions and reject a minimum of valid transitions. We designed our inference algorithm with these objectives. In the following, we verify the correctness and measure the completeness of inferred protocol grammars.

Applied to the IRC protocol, our inference algorithm produces the state machine illustrated on Figure 12.1. It contains eight states and accepts no invalid transitions, the inferred machine is correct according to the algorithm detailed in Section 12.2. Regarding its completeness, we executed a manual comparison of its state machine against the one obtained by means of the L^* algorithm. Our state machine accepts all the transitions traversed by normal users, *i.e.* that generates no protocol errors. However, some transitions inferred by L^* algorithm are missing in our model. They denote the error management process of the IRC implementation. For example, our algorithm did not infer the transition triggered by the *JOIN* symbol when the user is not yet authenticated. Indeed, the IRC protocol does not accept that a user joins a channel if he is not authenticated. This transition is a self-loop transition that returns no symbol. Indeed, all the transitions we missed are transitions triggered by sequence of symbols that are not accepted by the protocol implementation. All these transitions act similarly as they are all self-loop transitions that generates no output symbol. Thus by implementing a default strategy that does not answer when our state machine has no transition given the current state and the received symbol, we can ensure the completeness of our model.

Regarding the SAMBAv2 protocol, our approach inferred the state machine illustrated on Figure 12.2. Our evaluation of its correctness reveals that the protocol implementation accepts all the transitions accepted by the inferred state machine. For example, the produced state machine correctly denotes the SAMBA authentication schema that relies on a succession of *ComNegotiateRequest* and *SessionSetupRequest* symbols. It also inferred that initiating an *NMBSessionRequest* is not mandatory for the authentication process. Besides, our approach has successfully discovered that any manipulation of a file required a prior-emission of *TreeConnectRequest* symbol. However, the inferred state machine is not complete. Similarly to the IRC protocol, some transitions related to the emission of invalid sequences of symbols are missing in our result. Implementing a default strategy for these transitions is sufficient to address this issue. However, one additional transition is missing in the state machine inferred by our approach. This transition is related to the heartbeat mechanism proposed by the SAMBAv2 protocol. It is represented by a self-loop transition triggered by the *EchoRequest* symbol that answers with an *EchoResponse* symbol. This transition is accepted by every states of the protocol grammar when the user is authenticated. Our approach successfully inferred this transition except for one state. In this state, the inferred transition triggered by the *EchoRequest* symbol is not a self-state transition. Our approach wrongly replaced this transition with a transition that ends on a final state of the protocol, *i.e.* a state that accepts no other transition. This incompleteness is due to the fact that our merging algorithm failed to infer an equivalent ending state for this transition. Such case happens, when contextual information are not enough. Thus our merging algorithm created a new state to host this transition instead of identifying that the initial state was equivalent to the ending state. Despite this error, the inferred state machine infers all the transitions a normal user (*i.e.* conversely to a fuzzer for instance) would follows.

Finally, our inference algorithm returned the state machine illustrated on Figure 12.3 as the grammar of the botnet protocol. Made of eight states, this state machine describes an authentication path that goes from state 1 to state 2. Once authenticated, the bot master has access to various commands such as *TCPFlood* or *PScan*. However, the PBot implementation accepts no successive repetition of a command. This explains the complexity of its state machine and the lack of any self-loop transition. In addition to these commands, the bot master can also logout from the botnet by means of the *Logout* command. Our inference algorithm successfully inferred that this command can be triggered on every states reachable by the authenticated user. The evaluation of its correcteness shows that our algorithm inferred a correct state machine. Besides, the only transitions we missed are related to the emission of invalid sequence of symbols. For instance, our algorithm did not infer that no symbol was answered by the botnet when the authenticated user tries to re-authenticate himself after sending an *TCPFlood* command. Indeed, the implementation of a default strategy similarly to the one we proposed for the IRC protocol is enough to obtain a complete and correct state machine of the protocol grammar.

12.4.2 Comparing Inference Times

As described previously, our approach can be executed in parallel to reduce the inference time. To do so, we create an inference thread for each protocol action we identified. In the following, we compare the inference time required by our approach to obtain the state machine against the inference time required by the traditional L^* algorithm.

Table 12.3 details the timing and stealthiness metrics we measures from the traditional L^* and action-based L^* inference processes when applied on the IRC protocol. It shows that the traditional L^* algorithm returned the grammar of the protocol after 2.31 hours (8 341 seconds) of computation. In comparison, our approach only required 18 minutes (1 116 seconds) to complete if executed in parallel. Besides, the sequential execution of our algorithm would also requires a slightly reduced inference time than the traditional L^* . We assume this improvements comes from our reduction of the L^* complexity we described in Section 10.3. When executed in parallel, it represents a speedup of 7.4. In details, our approach relied on four parallel threads, each being assigned to the inference of an action of the IRC protocol. The first thread inferred the connection action state machine after 1 115 seconds. The second thread returned the state machine of the user management action after 932 seconds while the third thread took 801 seconds to infer the grammar of the channel management action. The last thread inferred the state machine of the disconnection action in 185 seconds. Finally, our merging algorithm returned the protocol state machine in less than one second. Combined to our analysis of the correctness and completeness of the obtained IRC grammar, these results confirm that our approach can significantly reduce the total inference time required to obtain a valid and almost complete protocol state machine when applied on the IRC protocol.

Protocol	**IRC**				
Algorithm	Traditional L^*	Action-based L^*			
Thread ID	1	1	2	3	4
Duration (sec.)	8 341	1 115	932	801	185
Ns	23 795	3 276	2 808	2 369	851
Ds	2.85	2.93	3.01	2.95	4.59
Nes	1 785	392	62	261	17
Des	0.21	0.2	0.06	0.32	0.07
Nq	4 650	700	539	450	213

Table 12.3 – Experimental results on the IRC protocol.

Regarding the SAMBAv2 protocol, our approach returned the grammar after 11 760 seconds representing 3.26 hours of computation. It denotes a speedup of 3.17 in comparison to the traditional L^* inference that took more than 10 hours (37 381 seconds) to compute. These results are detailed in Table 12.4. Similarly to the IRC inference process, we also measured the inference time required by each of our threads. Once more, its the action grammar having the largest number of states that took the more time to infer.

Finally, the traditional L^* algorithm inferred the Botnet protocol after 11 324 seconds of computation which represents 3.14 hours. As detailed on Table 12.5, our action-based algorithm inferred its state machine after 8 466 seconds (2.35 hours). In this case, the speed up factor brought by our algorithm is limited (1.34). This is due to the unbalanced distribution of states and symbols between the actions, *i.e.* the state machine of the commands action has much more states and

Protocol	**SAMBAv2**			
Algorithm	Traditional L^*	Action-based L^*		
Thread ID	1	1	2	3
Duration (sec.)	37 381	11 053	11 759	3 106
Ns	47 723	16 147	8 613	1 927
Ds	1.27	1.46	0.73	0.62
Nes	23 544	6 962	2 922	572
Des	0.62	0.63	0.25	0.18
Nq	5 166	1 715	829	124

Table 12.4 – Experimental results on the SAMBAv2 protocol.

symbols than the other actions. Indeed, the thread that inferred it took 8 466 seconds to complete while inferring the login and logout actions respectively required 1 997 and 1 000 seconds to complete.

Protocol	**PBot**			
Algorithm	Traditional L^*	Action-based L^*		
Thread ID	1	1	2	3
Duration (sec.)	11 324	1 000	1 997	8 466
Ns	8 360	756	1 495	6 258
Ds	0.73	0.75	0.74	0.73
Nes	4 628	433	838	3 582
Des	0.06	0.05	0.05	0.06
Nq	913	56	122	653

Table 12.5 – Experimental results on the PBot protocol.

Table 12.6 highlights the relationship that exists between the distribution of states and symbols across actions and the speed up factor offered by our approach. To measure the distribution of states and symbols among the actions, we rely on their variances. A lower variance indicates a better homogeneous distribution. Applied to our experiments, this measure shows that our approach offers the best speed-up factor in comparison to the L^* algorithm when the distribution of symbols is homogeneous among the actions. Our evaluation on the IRC protocol is an example of such homogeneous distribution. Conversely, our experiments on the SAMBAv2 and PBot suffer from an heterogeneous distribution of symbols among their actions.

12.4.3 Comparing Inference Stealth

In the following, we detail our results on the stealthiness comparison of our approach against the L^* algorithm. We rely on the metrics we proposed in Section 7.3.

For the IRC protocol, the traditional L^* algorithm triggered the emission of 4 650 queries representing 23 795 symbols sent to the implementation (*i.e.* an average of 5.11 symbols per query).

Protocol	IRC	SAMBAv2	PBot
Speed Up Factor	7.4	3.17	1.34
Number actions	4	3	3
Var(Number of Symbols per Action)	0.25	4.22	4.66

Table 12.6 – Obtained Speed Up factors compared against the distribution of states and symbols between actions.

Executed queries triggered the reception of 1 785 error symbols with an average density of 0.21 symbol sent per second. In comparison, our approach only required 1 902 queries to infer the protocol state machine. It denotes a decrease by 59% of the total number of implementation resets. This reduction also explains why our approach completed faster. Regarding the number of error symbols, our inference algorithm also generated 59% less of them with a total of 732 erroneous symbols, all threads combined. Despite the fact that 732 protocol mistakes can still be detected by anti-inference techniques that monitors all our threads, it shows that our solution is far more stealthy than the L^* algorithm. Besides, if we attach each thread of our inference algorithm to a different implementation of the protocol, we can consider that our algorithm only triggered a maximum of 392 protocol mistakes. In this situation, it denotes a decrease by 83% of the number of protocol mistakes sent to each protocol implementation.

These numbers are even more interesting if we optimize the stealthiness of our approach. As explained previously, our approach is faster than the traditional L^* algorithm mostly because we managed to execute our algorithm in parallel. Thus, we can voluntary reduce the inference speed by introducing a small break before emitting each symbol to the implementation. This way, we highly reduce the average density of sent symbols to each implementation. This solution allows our inference algorithm to be used on protected protocols and to obtain their grammar in a reasonable amount of time.

As described in Table 12.4, our approach also requires fewer queries than the traditional L^* algorithm to infer the SAMBA grammar, *i.e.* 59% less queries. Besides, if we sum the number of symbols sent by each of our threads, our inference algorithm sent 26 687 symbols to the protocol implementation. It shows a decrease by 44% in comparison to the 47 723 symbols sent by the L^* algorithm. Indeed, our inference algorithm requires less stimulation of the protocol implementation thus being stealthier. Similarly to the results obtained on the IRC protocol, our approach also generates fewer protocol mistakes to complete, *i.e.* 55% less protocol mistakes than L^* .

However, the results brought by our inference process when applied on the PBot protocol are less effective (see Table 12.5). For example, our approach required only 9% less queries to infer the state machine than the L^* algorithm. Moreover, the total number of erroneous symbols submitted as part of our algorithm is slightly higher than the number of erroneous symbols generated by the L^* algorithm. This increase is once more due to the heterogeneous distribution of symbols and states among the retained actions to infer the protocol. However, it should be noted that we executed our inference process in parallel on three different instances of PBot. Thus, we can compare the number of queries, of symbols and of erroneous symbols sent to the implementation by the L^*

algorithm against those generated by the worst thread of our inference. This comparison assumes that no distributed anti-inference technique is implemented in the botnet. In this case, our approach is stealthier than the L^* inference process. Indeed, each PBot instance receives fewer symbols and among them fewer erroneous symbols than the instance used with the L^* algorithm.

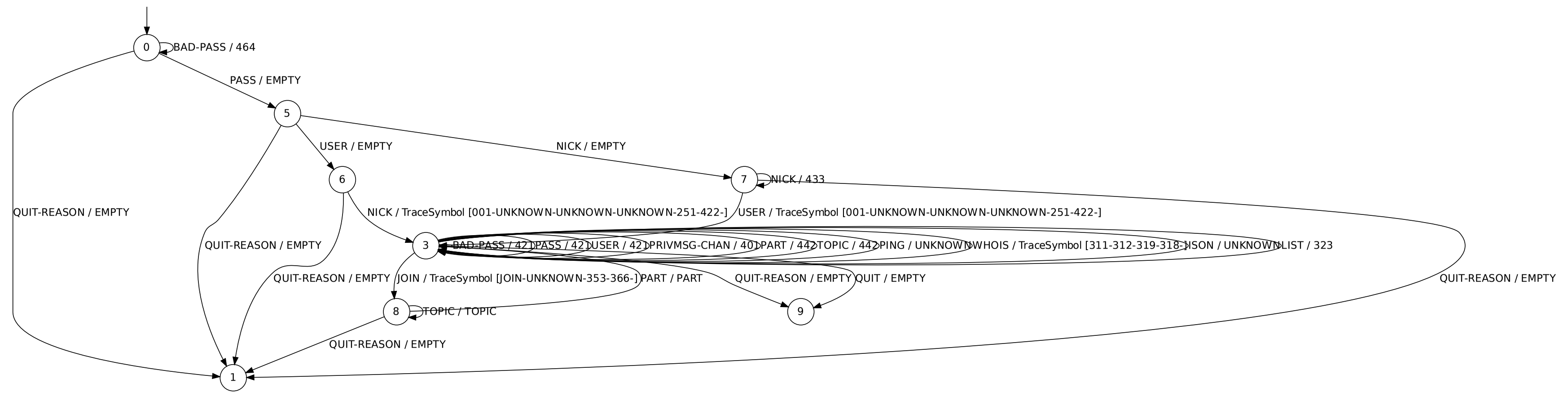


Figure 12.1 – Inferred state machine of the IRC protocol (self-state transitions triggering an empty symbol and reaction time labels are removed for sake of clarity) .

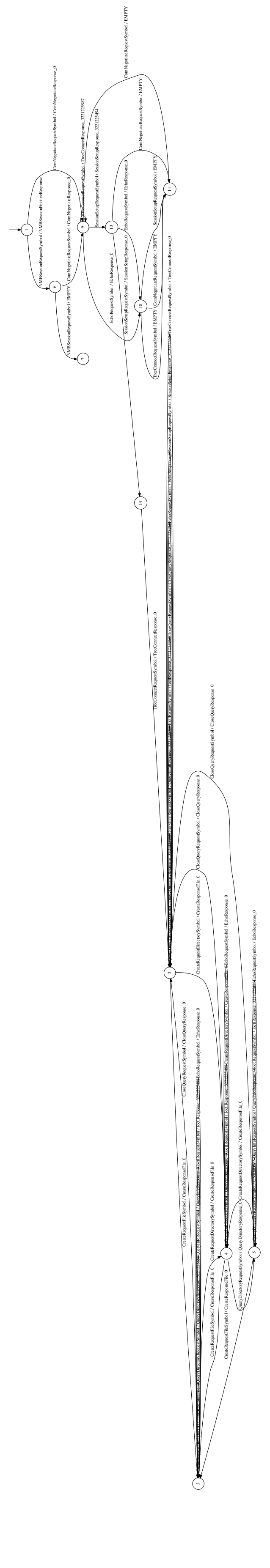


Figure 12.2 – Inferred state machine of the Samba protocol (self-state transitions triggering an empty symbol and reaction time labels are removed for sake of clarity).

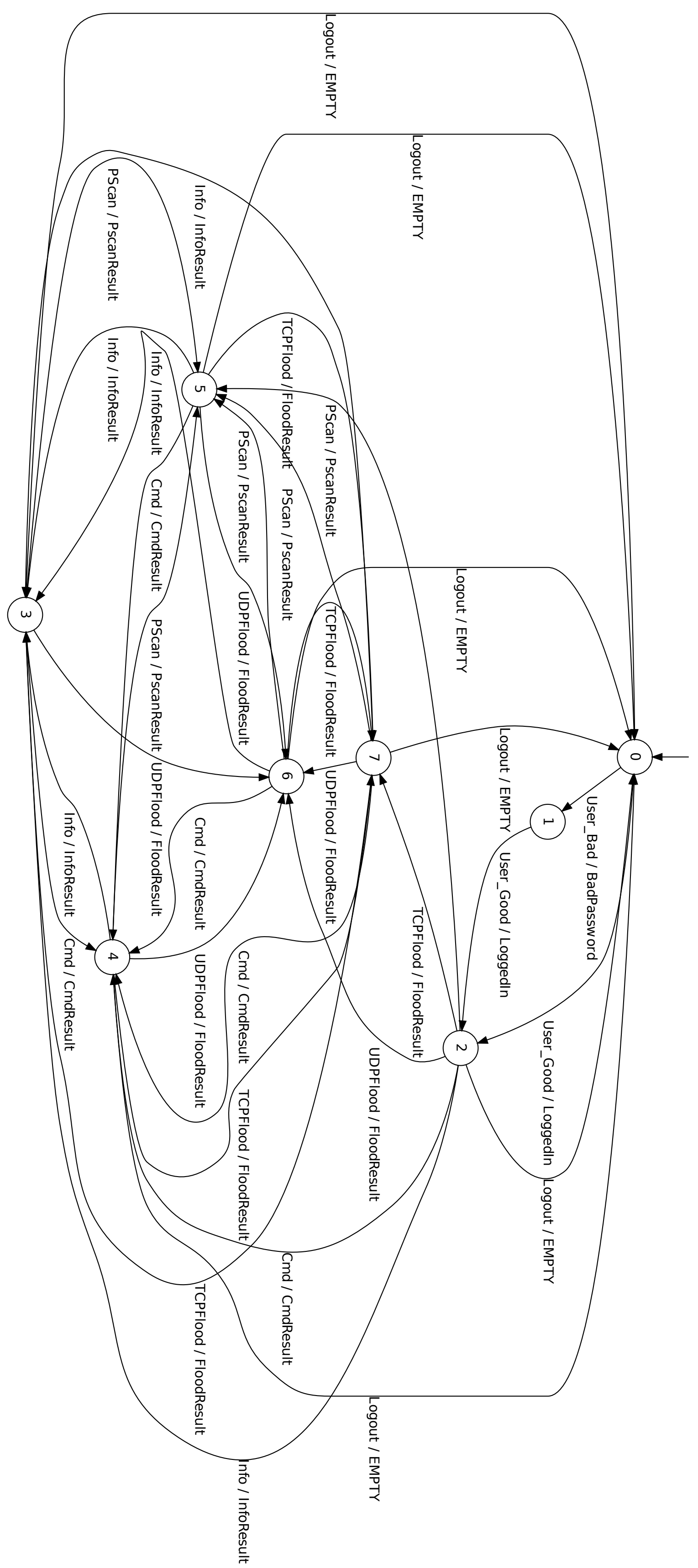


Figure 12.3 – Inferred state machine of the Botnet protocol v1 (self-state transitions triggering an empty symbol and reaction time labels are removed for sake of clarity).

Chapter 13

Conclusion on Grammar Inference

In this Chapter, we proposed an automated approach to infer the grammar of a protocol. Our approach combines a passive and an active approach to improve the stealthiness of the inference process. It also propose a solution to execute in parallel the inference of a protocol. To achieve this, we leverage semantic information to identify the different actions that are involved in the grammar of a protocol. We implemented our approach in a publicly available framework, and demonstrated its efficiency against three different protocols. Moreover, we compared our approach against the traditional L^* algorithm. The experimentation shows that it can infer a good approximation of the grammar of a protocol using fewer queries and symbols than traditional L^* . Our solution is also effective to reduce the inference time of large grammars.

Chapter 14

Conclusion

This chapter concludes our thesis. At first, Section 14.1 recalls the objectives set out in Section 1.3 and studies their achievements. We then propose improvements to our work in Section 14.2.

14.1 Results

In this thesis, we proposed an approach for the reverse engineering of a communication protocol with three main objectives: infer a precise, complete and correct model of an undocumented protocol (**objective 1**) while reducing the computation time (**objective 2**) and improving the stealthiness (**objective 3**) of the inference process in comparison to previous work.

To attain these objectives, our approach relies on original techniques that leverage semantic information to enhance both the inference of the protocol vocabulary and of its grammar. We implemented our approach in a publicly available framework, and demonstrated its efficiency against standard and unknown protocols.

In the first part of this thesis, we proposed an approach to infer the vocabulary of a protocol based on collected samples of communications. To achieve this, we conceived a fine-grained vocabulary model and a methodology that infers it. Our contributions mostly relies on our intuition that some semantic information can be collected along with communication traces to drive message clustering and alignment. We considered two different types of semantic information to achieve this: 1) the nature of the operations performed on the protocol implementation while messages are captured and 2) various contextual information such as timestamps or IP sources addresses. We extended both clustering and sequence alignment algorithms to leverage them. Furthermore, we also explored the delicate complexity-precision trade-off involved with the identification of complex linear and non-linear relationships that could participate in message definitions. Finally, we proposed a comparative study that relies on quantitative metrics to compare our solution against three state-of-the-work solutions we re-implemented.

Obtained results shows that our vocabulary inference solution returns better results than existing work in terms of model preciseness, completeness and correctness (**objective 1**). Besides, we believe that the use of a multi-step clustering algorithm and of a correlation based relationship identification solution reduces the overall complexity of the inference and therefore limits its computation time (**objective 2**). However, this objective is partially achieved as future works could be done to precisely measure it. Nevertheless, our solution relies on a trace-based approach which ensures its stealthiness (**objective 3**).

The second part of this thesis detailed our solution to infer the grammar of a protocol by means of a passive and active approach. We followed the same intuition than for vocabulary inference as we also relied on semantic information to reduce the inference time and improve its stealthiness. We have shown that semantic information can be used to split the large inference task into separate sub-tasks that can be executed in parallel. This decomposition of the protocol state machine reduces the theoretical complexity of the L^* algorithm while supporting its execution in parallel. Our combination of a passive and an active technique through the use of a cache and in our merging algorithm also reduces the stimulation of the targeted protocol implementation. Finally, our grammar model also covers the reaction time which is automatically inferred by our solution. This knowledge of the reaction time improves the completeness of our inferred model.

Regarding our objectives, we claim that our contributions in grammatical inference partially fulfilled the first objective as the technique we propose relies on heuristics that may lead to incorrect and/or incomplete results on some protocols. However, we detailed in Section 12.4.1 that this incompleteness can be addressed by introducing a default policy to model non-inferred transitions. Nevertheless, our experimentation confirmed that our approach can be used to infer a precise, correct and almost complete grammar of an unknown protocol (**objective 1**). Moreover, the results exposed in Chapter 12 shows that our solution requires fewer computation time (**objective 2**) while being stealthier (**objective 3**) than previous work.

14.2 Perspectives

The work on protocol reverse engineering is far from over, yet we believe this thesis proposes many improvements in this domain. Indeed, despite our efforts, the complete automation of protocol reverse engineering has not yet been reached. Completing the protocol model, implementing and testing new approaches would need to be tackled to ensure the improvements of protocol reverse engineering techniques and their wide adoption by security experts. Nevertheless, we believe that our work is one more step paving the way towards automated protocol RE. We identified some directions for future work, including the use of an active inference approach to infer unobserved protocol messages and the combination of trace-based and binary-based inference algorithms.

Similarly to the solution we retained to infer the grammar of a protocol, we could extend our vocabulary inference algorithm with an active algorithm. For example, an active algorithm could be used to improve the inference of the definition domains of each field by submitting messages with different values. Such approach could also be interesting to confirm inferred relationships.

Another interesting research path could be to combine our traced-based approach with binary-based RE techniques. Indeed, analyzing the construction of buffers could bring additional information that could improve our clustering and field discovery algorithms.

Exploring the automated protocol vulnerability assessment through the creation of smart-fuzzers is another future work. Such tool could be automatically generated based on the protocol model we inferred. It would introduce deviations in both the message formats and in the grammar of the protocol.

Nonetheless, we believe the most interesting and multifaceted venue of future work would be to support expert intervention into the inference process. We could adapt our algorithms to this and let the expert corrects, modifies or extends the inferred specifications to tune the remaining inference processes and improve their results.

List of Figures

Glossary of Accronyms

ABNF Augmented Backus-Naur Form. 16, 29, 36, 37

ASCII American Standard Code for Information Interchange. 17, 18, 23, 24, 37, 39–41, 48, 51, 62, 66, 84

ASN.1 Abstract Syntax Notation One. 29–33, 38

ATE Assurance TEsting. 10

ATM Asynchronous Transfer Mode. 56

AV Antivirus. 8, 11

BER Basic Encoding Rules. 32–34

BSC IBM Bi-SynC protocol. 15

CAP Common Alerting Protocol. 34

CAPS CSEG Assisted Product Service. 9

CC Common Criteria. 9, 10

CCITT Comité Consultatif International Téléphonique et Télégraphique. 29, 30

CDF Cumulative Distributed Function. 88

CER Canonical Encoding Rules. 32–34

CFG Context-Free Grammar. 27, 28

CR Carriage Return. 16, 17, 37

CRC Cyclic Redundancy Check. 20, 21, 52

CSG Context-Sensitive Grammar. 28

CSPN Certification de Sécurité de Premier Niveau. 9

DDOS Distributed Denial Of Service. 7, 56

DER Distinguished Encoding Rules. 32–34

DFA Deterministic Finite Automaton. 54, 55

DHCP Dynamic Host Configuration Protocol. 24, 52

DNS Domain Name Service. 6, 24, 25

EAP Extensible Authentication Protocol. 10

EFSM Extended Finite State Machine. 41

EPICS Experimental Physics and Industrial Control System. 18

EQ Equivalency Queries. 55

FDT Formal Description Techniques. 29

FIFO First-In First-Out. 40, 41

FPGA Programmable Gate Arrays. 101

FSM Finite State Machine. 51, 55, 84, 101

FTP File Transfer Protocol. 7, 18, 22, 74, 83–86, 88, 91, 102, 103

GSER Generic String Encoding Rules. 36

HIDS Host-based Intrusion Detection System. 11

HTTP Hypertext Transfer Protocol. 8, 16, 17, 24, 45, 47, 62

ICMP Internet Control Message Protocol. 22

ICS Industrial Constrol System. 16

ICT Information and Communications Technology. 7

IDL Interface Description Language. 37, 38

IDS Intrusion Detection System. 5, 8, 10

IEEE Institute of Electrical and Electronics Engineers. 15, 16

IETF Internet Engineering Task Force. 16, 36

IOC Input Output Controllers. 18

IP Internet Protocol. 6, 15, 18, 22–24, 34, 45–47, 52, 62, 64, 74, 85

IPsec Internet Protocol Security. 15

IRC Internet Relay Chat. 8, 18, 66, 75, 117, 118, 120–126

ISO International Standards Organization. 6, 16, 18, 40, 41

ITU International Telecommunication Union. 16, 30, 32–34

K-S Kolmogorov-Smirnov. 47

LAN Local Area Network. 6, 46

LBNL Lawrence Berkeley National Laboratory. 85

LDAP Lightweight Directory Access Protocol. 36

LF Line Feed. 16, 17, 37

LOTOS Language of Temporal Ordering Specification. 40, 41

MAN Metropolitan Area Network. 46

MBR Master Boot Record. 18

MQ Membership Queries. 55

MSC Message Sequence Chart. 40

MSNP Microsoft Notification Protocol. 7

MSS Maximum Segment Size. 47

MTU Maximum Transmission Unit. 46

NIDS Network Intrusion Detection System. 9–11

NTP Network Time Protocol. 7

NW Needleman & Wunsch. 49–51, 77, 78

OSI Open Systems Interconnection. 6, 21, 29, 30, 46

P2P Peer to Peer. 8, 10, 15, 19, 21, 84–86

PDML Packet Details Markup Language. 88

PDU Protocol Data Unit. 6, 45

PER Packed Encoding Rules. 32, 34, 36, 38, 40

PI Protocol Informatic project. 49

PIP P Internet Protocol. 62

PKCS Public Key Cryptographic Standards. 33

PLC Programmable Logic Controllers. 18

PLD Programmable Logic Devices. 101

PPC Pay-Per-Click. 7

PR Production Rules. 26, 27

ProtoBuf Protocol Buffer. 29, 37–39

PTA Prefix Tree Acceptor. 54, 55

PTP Precision Time Protocol. 7

RCP Rate Control Protocol. 7

RE Reverse Engineering. 8–13, 43, 59

RFC Request for Comments. 16, 37

ROC Receiver Operating Characteristic. 88–90

RSS Representatives Sequence of Symbols. 107, 109, 112

RSYNC Remote Synchronization Protocol. 7

SEO Search Engine Optimization. 7

SER Sequence of Events Recorder. 44

SIP Session Initiation Protocol. 36, 37

SIS Security of Information Systems. 9

SMB Server Message Block. 23, 62, 76, 84–86, 88, 91

SMM Symbolic Mealy Machine. 14

SMTP Simple Mail Transport Protocol. 18

SNMP Simple Network Management Protocol. 31, 32, 36

SSL Secure Sockets Layers. 10

SVAS State Variable Assignment Strategy. 66, 67, 71

TCP Transmission Control Protocol. 6, 15–18, 21, 22, 25, 45–47, 61, 67, 74

TLS Transport Layer Security. 10

TLV Type-Length-Value. 32–34

ToE Target of Evaluation. 9, 10

TPSN Time-sync Protocol for Sensor Networks. 7

UDP User Datagram Protocol. 19, 30, 74

UPGMA Unweighted Pair Group Method with Arithmetic mean. 49, 77–79, 84

URL Uniform Resource Locator. 16

UTC Universal Time Coordinated. 44

VoIP Voice over Internet Protocol. 83, 85, 86

W3C World Wide Web Consortium. 16

WAN Wide Area Network. 46

XER XML Encoding Rules. 32, 34, 36

XMPP Extensible Messaging and Presence Protoco. 7, 22

XOR eXclusive OR. 19

ZA ZeroAccess. 83, 85, 86, 91

Bibliography

Bibliography

[1] ITU-T Study Group 17. Abstract syntax notation one (asn.1) - specification of basic notation. Technical report, International Telecommunication Union, 2002.

[2] Fides Aarts, Bengt Jonsson, and Johan Uijen. Generating models of infinite-state communication protocols using regular inference with abstraction. In *Proceedings of the 22nd IFIP WG 6.1 international conference on Testing software and systems*, ICTSS'10, pages 188–204, Berlin, Heidelberg, 2010. Springer-Verlag.

[3] Ibrahim S. Abdullah and Daniel A. Menasce. Protocol specification and automatic implementation using xml and cbse. In *Proc of the International Conference on Communications, Internet and Infomation technology*, 2003.

[4] Kenji Aiko. New reverse engineering technique using api hooking and sysenter hooking, and capturing of cash card access. In *Black Hat Asia*, 2008.

[5] Mark Amos. An intuitive explanation of cw bandwidth. `http://www.w8ji.com/cw_bandwidth_described.htm`.

[6] D. Andriesse and H. Bos. An analysis of the zeus peer-to-peer protocol. Technical report, VU University Amsterdam, may 2013.

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

[8] Igor Anishchenko. Pb vs thrift vs avro. In *Lohika*, 2012.

[9] J. Antunes, N. Neves, and P. Verissimo. Reverse engineering of protocols from network traces. In *Reverse Engineering (WCRE), 2011 18th Working Conference on*, pages 169 –178, oct. 2011.

[10] P. Ashar, S. Devadas, and A. R. Newton. Optimum and heuristic algorithms for an approach to finite state machine decomposition. *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, 10(3):296–310, nov 2006.

[11] Pranav Ashar, Srinivas Devadas, and A. Richard Newton. A unified approach to the decomposition and re-decomposition of sequential machines. In *Proceedings of the 27th ACM/IEEE Design Automation Conference*, DAC '90, pages 601–606, New York, NY, USA, 1990. ACM.

[12] The International MIDI Association. Standard midi-file format spec. 1.1. Technical report, The International MIDI Association, 1999.

[13] Paul Barford and Vinod Yegneswaran. An inside look at botnets. In *Malware Detection*, volume 27 of *Advances in Information Security*. Springer US, 2007.

[14] Marshall A. Beddoe. Network protocol analysis using bioinformatics algorithms. In *Toorcon*, 2004.

[15] P.L.T. Berg and T. Berg. *Structure in Language: A Dynamic Perspective*. Routledge Studies in Linguistics. Taylor & Francis, 2008.

[16] Therese Berg, Bengt Jonsson, Martin Leucker, and Mayank Saksena. Insights to angluin's learning. *Electron. Notes Theor. Comput. Sci.*, 118:3–18, February 2005.

[17] Therese Berg, Bengt Jonsson, and Siavash Soleimanifard. Inferring compact models of communication protocol entities. In *Leveraging Applications of Formal Methods, Verification, and Validation: Part I*, number 6415 in Lecture Notes in Computer Science, pages 658–672, 2010.

[18] Matt Bishop, Rick Crawford, Bhume Bhumiratana, Lisa Clark, and Karl Levitt. Some problems in sanitizing network data. In *15th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises*, pages 307–312, 2006.

[19] Gregor Bochmann. Protocol specification for osi. *Comput. Netw. ISDN Syst.*, 18(3):167–184, apr 1990.

[20] Therese Bohlin and Bengt Jonsson. Regular inference for communication protocol entities. Technical Report 2008-024, Uppsala University, Computer Systems, 2008.

[21] P. Borgnat, G. Dewaele, K. Fukuda, P. Abry, and K. Cho. Seven years and one day: Sketching the evolution of internet traffic. In *INFOCOM 2009, IEEE*, pages 711–719, 2009.

[22] Georges Bossert and Frédéric Guihéry. The future of protocol reversing and simulation applied on zeroaccess botnet. In *29C3: 29th Chaos Communication Congress*, 2012.

[23] Georges Bossert and Frederic Guihery. Security evaluation of communication protocols in cc. In *ICCC 2012*, 2012.

[24] Georges Bossert, Frederic Guihery, and Guillaume Hiet. Towards automated protocol reverse engineering using semantic information. In *ASIACCS*, 2014.

[25] Georges Bossert, Guillaume Hiet, and Thibaut Henin. Modelling to simulate botnet command and control protocols for the evaluation of network intrusion detection systems. In *Conference on Network and Information Systems Security (SAR-SSI)*, pages 1 –8, may 2011.

[26] Georges Bossert and Dimitri Kirchner. How to play hooker. In *SSTIC*, 2014.

[27] Juan Caballero, Pongsin Poosankam, Christian Kreibich, and Dawn Song. Dispatcher: enabling active botnet infiltration using automatic protocol reverse-engineering. In *Proceedings of CCS*, 2009.

[28] Juan Caballero and Dawn Song. Automatic protocol reverse-engineering: Message format extraction and field semantics inference. *Comput. Netw.*, 57(2):451–474, feb 2013.

[29] Juan Caballero, Heng Yin, Zhenkai Liang, and Dawn Song. Polyglot: Automatic extraction of protocol format using dynamic binary analysis. In *Proceedings of CCS*, 2007.

[30] J.D. Case, M. Fedor, M.L. Schoffstall, and J. Davin. Simple Network Management Protocol (SNMP). RFC 1157 (Historic), May 1990.

[31] Ana Cavalli, Cyril Grepet, Stéphane Maag, and Vincent Tortajada. A validation model for the dsr protocol. In *Proceedings of the 24th International Conference on Distributed Computing Systems Workshops - W7: EC (ICDCSW'04) - Volume 7*, ICDCSW '04, pages 768–773, Washington, DC, USA, 2004. IEEE Computer Society.

[32] CERTA. Certa-2004-ale-003 - propagation du ver phatbot. Technical report, S.G.D.S.N. Agence nationale de la sécurité des systèmes d'information, 2004.

[33] I.M. Chakravarti, R.G. Laha, and J. Roy. *Handbook of methods of applied statistics*. Number vol. 1 in Wiley series in probability and mathematical statistics. Wiley, 1967.

[34] Marco Chiesa, Luca Cittadini, Giuseppe Di Battista, Laurent Vanbever, and Stefano Vissicchio. Using routers to build logic circuits: How powerful is bgp? In *ICNP*, 2013.

[35] Chia Yuan Cho, Domagoj Babić, Eui Chul Richard Shin, and Dawn Song. Inference and analysis of formal models of botnet command and control protocols. In *Proceedings of the 17th ACM conference on Computer and communications security*, CCS '10, pages 426–439, New York, NY, USA, 2010. ACM.

[36] Hyunsang Choi, Heejo Lee, and Hyogon Kim. Botgad: detecting botnets by capturing group activities in network traffic. In *Proceedings of the Fourth International ICST Conference on COMmunication System softWAre and middlewaRE*, COMSWARE '09, pages 2:1–2:8, New York, NY, USA, 2009. ACM.

[37] N. Chomsky. *Syntactic Structures*. Mouton classic. Bod Third Party Titles, 2002.

[38] Noam Chomsky. Three models for the description of language. *Information Theory, IRE Transactions on*, 2(3):113–124, sep 1956.

[39] T. S. Chow. Testing software design modeled by finite-state machines. *IEEE Trans. Softw. Eng.*, 4(3):178–187, may 1978.

[40] Jurriaan Bremer Alessandro Tanasi Claudio Guarnieri, Mark Schloesser. Cuckoo sandbox - open source automated malware analysis. In *Black Hat USA*, 2013.

[41] Paolo Milani Comparetti, Gilbert Wondracek, Christopher Kruegel, and Engin Kirda. Prospex: Protocol specification extraction. In *Proceedings of SSP*, 2009.

[42] D. Crocker and P. Overell. Augmented BNF for Syntax Specifications: ABNF. RFC 5234 (INTERNET STANDARD), January 2008.

[43] Weidong Cui. Discoverer: Automatic protocol reverse engineering from network traces. In *Proceedings of USENIX Security Symposium*, 2007.

[44] Weidong Cui, Vern Paxson, Nicholas C. Weaver, and Y H. Katz. Protocol-independent adaptive replay of application dialog. In *In The 13th Annual Network and Distributed System Security Symposium (NDSS*, 2006.

[45] S. Devadas and A. R. Newton. Decomposition and factorization of sequential finite state machines. *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, 8(11):1206–1217, nov 2006.

[46] S. Djiev. Industrial networks for communication and control. Technical report, TU-Sofia Publ. House, 2003.

[47] Laurent Dolhi. *Validation of Communications Systems with SDl.* TransMeth Sud-Ouest, 2003.

[48] M. Franz E.J. Byres and D. Miller. The use of attack trees in assessing vulnerabilities in scada systems. In *International Infrastructure Survivability Workshop (IISW'04)*, 2004.

[49] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard), June 1999. Updated by RFCs 2817, 5785, 6266, 6585.

[50] Fortinet. Anatomy of a botnet. Technical report, Fortinet, 2013.

[51] P. Francis. Pip Near-term Architecture. RFC 1621 (Informational), May 1994.

[52] Jay Freeman. Hacking a closed ecosystem. In *O'Reilly Android Open Conference*, 2011.

[53] Susumu Fujiwara, Gregor von Bochmann, Ferhat Khendek, Mokhtar Amalou, and Abderrazak Ghedamsi. Test selection based on finite state models. *IEEE Trans. Softw. Eng.*, 17(6):591–603, jun 1991.

[54] Ullas Gargi. Consumer media capture: Time-based analysis and event clustering. Technical report, HP Laboratories Palo Alto, aug 2003.

[55] E Mark Gold. Complexity of automaton identification from given data. *Information and Control*, 37(3):302 – 320, 1978.

[56] Sergey Golovanov and Igo Soumenkov. Tdl4-top bot, June 2011.

[57] Guofei Gu, Phillip Porras, Vinod Yegneswaran, Martin Fong, and Wenke Lee. Bothunter: detecting malware infection through ids-driven dialog correlation. In *Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium*, pages 12:1–12:16, Berkeley, CA, USA, 2007. USENIX Association.

[58] M. Handley, H. Schulzrinne, E. Schooler, and J. Rosenberg. SIP: Session Initiation Protocol. RFC 2543 (Proposed Standard), March 1999. Obsoleted by RFCs 3261, 3262, 3263, 3264, 3265.

[59] D. Harel and P.S. Thiagarajan. *UML for Real: Design of Embedded Real-time Systems*, chapter Message Sequence Charts, page 1. Kluwer Academic Publishers, 2003.

[60] Juris Hartmanis. *Algebraic Structure Theory of Sequential Machines.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1966.

[61] Zafar Hasan and Maciej J. Ciesielski. Decomposition and functional verification of fsms. Technical report, Department of Electrical & Computer Engineering, University of Massachusetts, 1998.

[62] P. Hethmon. Extensions to FTP. RFC 3659 (Proposed Standard), March 2007.

[63] C.D. Higuera. *Grammatical inference: learning automata and grammars.* Cambridge University Press, 2010.

[64] Charles A.R. Hoare. *Communicating sequential processes*. Prentice-Hall, Inc., 1985.

[65] Gerard J. Holzmann. *Design and validation of computer protocols*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1991.

[66] Fraser Howard. Exploring the blackhole exploit kit. Technical report, SophosLabs, UK, march 2012.

[67] Yating Hsu, Guoqiang Shu, and D. Lee. A model-based approach to security flaw detection of network protocol implementations. In *Network Protocols, 2008. ICNP 2008. IEEE International Conference on*, pages 114–123, 2008.

[68] IBM. General information - binary synchronous communications. Technical report, IBM Systems Development Division, 1970.

[69] Open Systems Interconnection. Iso 8807:1989: Lotos – a formal description technique based on the temporal ordering of observational behaviour. Technical report, International Standards Organization.

[70] Muhammad Naeem IRFAN. *Analysis and optimization of software model inference algorithms*. PhD thesis, Laboratoire d'Informatique de Grenoble, September 2012.

[71] ISO. Information processing systems – OSI reference model, international standards organization. Technical Report 7498, ISO, October 1984.

[72] ITU-T. Information technology — asn.1 encoding rules — specification of basic encoding rules (ber), canonical encoding rules (cer), and distinguished encoding rules (der). Technical report, International Telecommunication Union, 2002.

[73] ITU-T. Formal description techniques (fdt) message sequence chart (msc). Technical report, ITU-T Z.120, 2011.

[74] C. Kalt. Internet Relay Chat: Architecture. RFC 2810 (Informational), April 2000.

[75] Michael Kende. Global internet report 2014. Technical report, Internet Society, 2014.

[76] S. Kent. Privacy Enhancement for Internet Electronic Mail: Part II: Certificate-Based Key Management. RFC 1422 (Historic), February 1993.

[77] S. Kent and K. Seo. Security Architecture for the Internet Protocol. RFC 4301 (Proposed Standard), December 2005. Updated by RFC 6040.

[78] S.C. Kleene. *Representation of Events in Nerve Nets and Finite Automata*. Memorandum (Rand Corporation). Rand Corporation, 1951.

[79] J. Klensin. Simple Mail Transfer Protocol. RFC 5321 (Draft Standard), October 2008.

[80] Hartmut Konig. *Protocol Engineering*. Springer, 2012.

[81] Chakravarty Tridib Koopman Philip. Analysis of the train communication network protocol error detection capabilities. Working paper, Carnegie Mellon University, 2001.

[82] Tammo Krueger, Hugo Gascon, Nicole Krämer, and Konrad Rieck. Learning stateful models for network honeypots. In *Proceedings of the 5th ACM workshop on Security and artificial intelligence*, 2012.

[83] Tammo Krueger, Nicole Kramer, and Konrad Rieck. Asap: automatic semantics-aware analysis of network payloads. In *Proceedings of ECML/PKDD*, 2011.

[84] Craig Labovitz, Scott Iekel-Johnson, Danny McPherson, Jon Oberheide, and Farnam Jahanian. Internet inter-domain traffic. *SIGCOMM Comput. Commun. Rev.*, 41(4):–, aug 2010.

[85] Kevin J. Lang. Faster algorithms for finding minimal consistent dfas. Technical report, NEC Research Institute, 4 Independence Way Princeton, NJ 08540, December 1999.

[86] Daniel D. Lee and H. Sebastian Seung. Algorithms for non-negative matrix factorization. In *NIPS*. MIT Press, 2000.

[87] S. Legg. Generic String Encoding Rules (GSER) for ASN.1 Types. RFC 3641 (Proposed Standard), October 2003. Updated by RFC 4792.

[88] Corrado Leita, Ken Mermoud, and Marc Dacier. Scriptgen: an automated script generation tool for honeyd. In *Proceedings of ACSAC*, 2005.

[89] Zhiqiang Lin, Xuxian Jiang, Dongyan Xu, and Xiangyu Zhang. Automatic protocol format reverse engineering through context-aware monitored execution. In *IN 15TH SYMPOSIUM ON NETWORK AND DISTRIBUTED SYSTEM SECURITY (NDSS*, 2008.

[90] L. Logrippo, M. Faci, and M. Haj-Hussein. An introduction to lotos: learning by examples. *Comput. Netw. ISDN Syst.*, 23(5):325–342, feb 1992.

[91] G.M. Lundy and C. Basaran. Automated generation of protocol test sequences from formal specifications. In *Network Protocols, 1994. Proceedings., 1994 International Conference on*, pages 72–79, Oct 1994.

[92] Song Luo and Gerald A. Marin. Modeling networking protocols to test intrusion detection systems. In *Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks*, LCN '04, pages 774–775, Washington, DC, USA, 2004. IEEE Computer Society.

[93] Eric Madelaine and Didier Vergamini. Specification and verification of a sliding window protocol in lotos. In *Formal Description Techniques, IV, volume C-2 of IFIP Transactions. Elsevier Science Publishers B.V. (North-Holland*, pages 495–510. North-Holland, 1991.

[94] Kevin McNamee. Malware analysis report - new c&c protocol for zeroacess/siref. Technical report, Kindsight Security Lab, 2012.

[95] R. Milner. *Communication and concurrency*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[96] J. C. Mogul, R. Fielding, J. Gettys, and H. Frystyk. Use and Interpretation of HTTP Version Numbers. RFC 2145 (Informational), May 1997.

[97] Saul B. Needleman and Christian D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48(3):443 – 453, 1970.

[98] Oliver Niese. *An Integrated Approach to Testing Complex Systems*. PhD thesis, Erlangung des Grades eines Doktors der Naturwissenschaften der Universitat Dortmund am Fachbereich Informatik, 2003.

[99] Vaibhav Nivargi, Mayukh Bhaowal, and Teddy Lee. Machine learning based botnet detection. Technical report, CS229, Standford, 2006.

[100] ODVA. Devicenet: Technical overview. Technical report, Open DeviceNet, Vendor Association, Inc., 2004.

[101] William Ogden. A helpful result for proving inherent ambiguity. *Mathematical systems theory*, 2(3):191–194, 1968.

[102] J. Oikarinen and D. Reed. Internet Relay Chat Protocol. RFC 1459 (Experimental), May 1993. Updated by RFCs 2810, 2811, 2812, 2813.

[103] OSI. Iso/iec 9074: Estelle - a formal description technique based on an extended state transition model. Technical report, Internationnal Organisation for Standardisation, 1989.

[104] Ruoming Pang and Vern Paxson. A high-level programming environment for packet trace anonymization and transformation. In *SIGCOMM '03: Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 339–351, New York, NY, USA, 2003. ACM.

[105] C.P. Pfleeger. State reduction in incompletely specified finite-state machines. *Computers, IEEE Transactions on*, C-22(12):1099–1102, 1973.

[106] J. Postel. User Datagram Protocol. RFC 768 (INTERNET STANDARD), August 1980.

[107] J. Postel. Internet Control Message Protocol. RFC 792 (INTERNET STANDARD), September 1981. Updated by RFCs 950, 4884, 6633, 6918.

[108] J. Postel. Internet Protocol. RFC 791 (INTERNET STANDARD), September 1981. Updated by RFCs 1349, 2474, 6864.

[109] J. Postel. Transmission Control Protocol. RFC 793 (INTERNET STANDARD), September 1981. Updated by RFCs 1122, 3168, 6093, 6528.

[110] J. Postel. Simple Mail Transfer Protocol. RFC 821 (INTERNET STANDARD), August 1982. Obsoleted by RFC 2821.

[111] Harald Raffelt, Bernhard Steffen, and Therese Berg. Learnlib: a library for automata learning and experimentation. In *Proceedings of the 10th international workshop on Formal methods for industrial critical systems*, FMICS '05, pages 62–71, New York, NY, USA, 2005. ACM.

[112] David N. Reshef, Yakir A. Reshef, Hilary K. Finucane, Sharon R. Grossman, Gilean McVean, Peter J. Turnbaugh, Eric S. Lander, Michael Mitzenmacher, and Pardis C. Sabeti. Detecting novel associations in large data sets. *Science*, 334(6062):1518–1524, 2011.

[113] P. Resnick. Internet Message Format. RFC 2822 (Proposed Standard), April 2001. Obsoleted by RFC 5322, updated by RFCs 5335, 5336.

[114] P. Resnick. Internet Message Format. RFC 5322 (Draft Standard), October 2008. Updated by RFC 6854.

[115] Konrad Rieck, Christian Wressnegger, and Alexander Bikadorov. Sally: A tool for embedding strings in vector spaces. *Journal of Machine Learning Research*, 2012.

[116] Eugene Rodionov and Aleksandr Matrosov. The evolution of tdl: conquering x64. Technical report, ESET, 2011.

[117] Martin Roesch. Snort - lightweight intrusion detection for networks. In *Proceedings of the USENIX LISA'99 conference*, pages 229–238, Seattle, WA, November 1999.

[118] E.C. Rosen. Vulnerabilities of network control protocols: An example. RFC 789, July 1981.

[119] P. Saint-Andre. Extensible Messaging and Presence Protocol (XMPP): Instant Messaging and Presence. RFC 6121 (Proposed Standard), March 2011.

[120] Brahima Sanou. Ict facts and figures. Technical report, ICT - International Telecommunication Union, 2013.

[121] Thilo Sauter. The three generations of field-level networks - evolution and compatibility issues. *IEEE Transactions on Industrial Electronics*, 57, November 2010.

[122] Telecommunication Standardization sector of ITU. Specification and description language (sdl). Technical report, ITU, 1988.

[123] Jarrad Shearer. Trojan.zeroaccess threat report. Technical report, Symantec, 2011.

[124] Sharon Shoham, Eran Yahav, Stephen Fink, and Marco Pistoia. Static specification mining using automata-based abstractions. In *Proceedings of the 2007 international symposium on Software testing and analysis*, ISSTA '07, pages 174–184, New York, NY, USA, 2007. ACM.

[125] Anas Showk, David Szczesny, Shadi Traboulsi, Irv Badr, Elizabeth Gonzalez, and Attila Bilgic. Modeling lte protocol for mobile terminals using a formal description technique. In *Proceedings of the 14th international SDL conference on Design for motes and mobiles*, SDL'09, pages 222–238, Berlin, Heidelberg, 2009. Springer-Verlag.

[126] Amichai Shulman. The untold tale of database communication protocol vulnerabilities. In *BlackHat*, 2007.

[127] R. R. Sokal and C. D. Michener. A statistical method for evaluating systematic relationships. *University of Kansas Scientific Bulletin*, 28:1409–1438, 1958.

[128] Adi Sosnovich, Orna Grumberg, and Gabi Nakibly. Finding security vulnerabilities in a network protocol using parameterized systems. In *Computer Aided Verification*, volume 8044 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2013.

[129] Guenther Starnberger, Christopher Kruegel, and Engin Kirda. Overbot: A botnet protocol based on kademlia. In *Proceedings of the 4th International Conference on Security and Privacy in Communication Netowrks*, SecureComm '08, pages 13:1–13:9, New York, NY, USA, 2008. ACM.

[130] Bernhard Steffen, Falk Howar, and Maik Merten. Introduction to active automata learning from a practical perspective. In *Formal Methods for Eternal Networked Software Systems*, volume 6659 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2011.

[131] Joe Stewart. Inside the storm: Protocols and encryption of the storm botnet. In *Black Hat USA*, 2008.

[132] Apostolos Syropoulos. Mathematics of multisets. In *Proceedings of the Workshop on Multiset Processing: Multiset Processing, Mathematical, Computer Science, and Molecular Computing Points of View*, WMP '00, pages 347–358, London, UK, UK, 2001. Springer-Verlag.

[133] Dariusz Tasak. Specification and validation of q.2931 atm signaling protocol using estelle. Master's thesis, School of Computer Science McGill University, Montreal, September 1997.

[134] OASIS Emergency Management TC. Common alerting protocol version 1.2. Technical report, OASIS, 2010.

[135] Isabelle Tellier. Learning recursive automata from positive examples. *Revue d'Intelligence Artificielle*, 20(6):775–804, 2006.

[136] Gilou Tenebro. W32.waledac - threat analysis. Technical report, Symantec, 2009.

[137] Jay Turla. Analysis on pbot – a php irc bot that has malicious functions. Technical report, INFOSEC Institute, 2012.

[138] Yipeng Wang, Xiaochun Yun, M. Zubair Shafiq, Liyan Wang, Alex X. Liu, Zhibin Zhang, Danfeng(Daphne) Yao, Yongzheng Zhang, and Li Guo. A semantics aware approach to automated reverse engineering unknown protocols. In *Proceedings of ICNP*, 2012.

[139] Yipeng Wang, Zhibin Zhang, Danfeng Daphne Yao, Buyun Qu, and Li Guo. Inferring protocol state machine from network traces: a probabilistic approach. In *Proceedings of ACNS*, 2011.

[140] Zhi Wang, Xuxian Jiang, Weidong Cui, Xinyuan Wang, and Mike Grace. Reformat: Automatic reverse engineering of encrypted messages. In *Proceedings of the 14th European Conference on Research in Computer Security*, ESORICS'09, pages 200–215, Berlin, Heidelberg, 2009. Springer-Verlag.

[141] Youngjoon Won, R. Fontugne, K. Cho, H. Esaki, and K. Fukuda. Nine years of observing traffic anomalies: Trending analysis in backbone networks. In *Integrated Network Management (IM 2013), 2013 IFIP/IEEE International Symposium on*, pages 636–642, 2013.

[142] Gilbert Wondracek, Paolo Milani Comparetti, Christopher Kruegel, and Engin Kirda. Automatic network protocol analysis. In *Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS'08*, 2008.

[143] Peter Wurzinger, Leyla Bilge, Thorsten Holz, Jan Goebel, Christopher Kruegel, and Engin Kirda. Automatically generating models for botnet detection. In *Proceedings of the 14th European conference on Research in computer security*, ESORICS'09, pages 232–249, Berlin, Heidelberg, 2009. Springer-Verlag.

[144] Tao Xie. Software component protocol inference. Technical report, Department of Computer Science and Engineering, University of Washington, 2003.

[145] Tom Yeh, Tsung-Hsiang Chang, and Robert C. Miller. Sikuli: Using gui screenshots for search and automation. In *Proceedings of the 22Nd Annual ACM Symposium on User Interface Software and Technology*, UIST '09, pages 183–192, New York, NY, USA, 2009. ACM.

[146] K. Zeilenga. Lightweight Directory Access Protocol (LDAP): Technical Specification Road Map. RFC 4510 (Proposed Standard), June 2006.

Author’s publication

Publications of the Author

14.3 International Peer Reviewed Publications

— **Security Evaluation of Communication Protocols in Common Criteria**, Georges Bossert and Frédéric Guihéry, *International Common Criteria Conference* - September 2012

— **Towards Automated Protocol Reverse Engineering Using Semantic Information**, Georges Bossert, Frédéric Guihéry and Guillaume Hiet, *9th ACM Symposium on Information, Computer and Communication Security* - June 2014

14.4 National Peer-Reviewed Publications

— **Modelling to Simulate Botnet Command and Control Protocols for the Evaluation of Network Intrusion Detection Systems**, Georges Bossert, Guillaume Hiet and Thibaut Hénin, *Conference on Network and Information Systems Security (SAR-SSI)* - 2011

— **Netzob : un outil pour la rétro-conception de protocoles de communication**, Georges Bossert, Frédéric Guihéry and Guillaume Hiet, *Symposium sur la Sécurité des technologies de l'Information et des Communications (SSTIC)* - 2012

14.5 International Security Conferences

— **Reverse and Simulate your Enemy Botnet C&C**, Georges Bossert and Frédéric Guihéry, *Black Hat - Abu Dhabi* - 2012

— **The future of protocol reversing and simulation applied on ZeroAccess botnet**, Georges Bossert and Frédéric Guihéry, *29C3: 29th Chaos Communication Congress* - 2012

14.6 Article

— **Vivisection de protocoles avec Netzob**, Georges Bossert and Frédéric Guihéry, *MISC HS7* - Les éditions diamands, 2013

L’Ange me coupa la parole : Quand cesseras-tu, puceron orgueilleux et éphémère, de toujours t’agiter, de discutailler et d’ergoter ? Quand la Nuit, fraternelle et sûre, impérieuse et souveraine, s’apprête à descendre, il n’est plus temps de bavarder encore. Au seuil de l’Éternité, fais enfin silence et, dans l’obscurité qui d’heure en heure maintenant s’épaissit, au lieu de parler, écoute... : “Ici, le rameur enlève les avirons et amarre sans bruit dans les roseaux, avant de s’éloigner, une barque prêtée.”
Puis l’Ange disparut, sans bruit, dans la nuit, me laissant seul, au moins en apparence.

Théodore Monod, L’émeraude des Garamantes