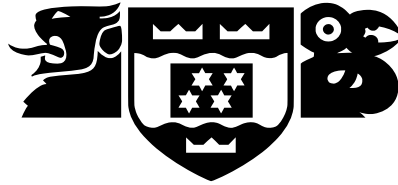


VICTORIA UNIVERSITY OF WELLINGTON
Te Whare Wānanga o te Ūpoko o te Ika a Māui



School of Engineering and Computer Science
Te Kura Mātai Pūkaha, Pūrorohiko

PO Box 600
Wellington
New Zealand

Tel: +64 4 463 5341
Fax: +64 4 463 5045
Internet: office@ecs.vuw.ac.nz

Finite State Automata Representation of Protocol Symbols from Network Traces

Lewel Lenz Nerit

Supervisors: Dr David Streader and Dr Ian Welch

26 February 2016

Submitted in partial fulfilment of the requirements for
Master of Computer Science.

Abstract

An increasing number of network applications are implementing proprietary or unknown protocols. The proprietary protocols do not have their specifications available to the public. Analysing and understanding the functionality of these proprietary protocols is an important aspect to discovering security loop holes in the implementation of these protocols. By inferring and modelling a protocol using a state machine, we may be able to understand its behaviour. In this project, we developed an open-source automatic state machine inference tool called kTail-PSM. We use an open-source tool called Netzob to extract protocol specifications (symbols). The protocol symbols extracted using Netzob are then fed into the kTail-PSM tool to infer the protocol state machine. We evaluate our tool with one text-based protocol (SMTP) and one binary protocol (TCP), finding our tool was able to generate a high quality state machine for each protocol. We compared our protocol state machines against the reference protocol state machines specified in the RFC, also found that our protocol state machines generated by our tool showed more detail than the ones defined in the RFC specifications.

Acknowledgements

I would like to take this opportunity with great pleasure to thank all of the people who provided time and resources to support this project. In particular, I would like to thank Dr Ian Welch and Dr David Streader, for supervising the project and for supporting and giving me insights into the understanding of Protocol Reverse Engineering concepts and the formal modelling concepts. Special thanks to Truong Trung for the support and continuous discussions towards my project. I would also like to extend my special thanks to Matthew Stevens for proofreading my project report.

Furthermore, I would also like to thank my parents, late Nerit Warom and Rup Warom who provided me with continuous support and their prayers. Also my special thanks to wife Sylvia, three sons Quizta'Lenz, Mangii'lenz, Qualdri'Lenz and also to uncle Danat Tinapit for their patience and continuous support they have provided throughout the two years of my study. Finally but not the least, I would like to thank the New Zealand Government through Ministry of Foreign Affairs and Trade (MFAT) for giving me the prestigious scholarship opportunity to complete Master of Computer Science degree at Victoria University of Wellington.

Contents

1	Introduction	1
2	Background and Literature Review	3
2.1	Protocol Reverse Engineering (PRE) Prerequisites	3
2.1.1	Network Trace-based Input	3
2.1.2	Program-based Input	3
2.1.3	Target Protocol Encoding	4
2.2	Protocol Specification	4
2.2.1	Protocol Vocabulary	4
2.2.2	Protocol Grammar	4
2.2.3	Finite State Machines (FSM)	4
2.3	Protocol Specification Extraction Methods	5
2.3.1	Grammatical Inference	5
2.3.2	Statistical Analysis	5
2.3.3	Keyword Identification	6
2.3.4	Static and Dynamic Analysis	6
2.3.5	Other Approaches	7
2.4	Classification of Automatic PRE Tools	7
2.4.1	Protocol Message Format/Syntax Extraction Tools	7
2.4.2	Protocol State Machine Modelling Tools	9
2.4.3	Availability of automatic PRE Tools	10
2.5	k-Tail Algorithm Overview	10
2.6	Using FSM for validation and Parsing Tokens	11
3	Design	13
3.1	Engineering Constraints	13
3.2	Design Goals	13
3.3	Design Approach	14
3.3.1	Using Netzob	14
3.3.2	Using Wireshark	16
3.3.3	k-Tail Protocol State Machine(PSM) Tool	17
3.3.4	Finite State Automata (FSA) Representation of Protocol Symbols . . .	18
3.3.5	Defining States and Transitions from a given Sequence of Symbols . .	18
4	Implementation	19
4.1	kTail-PSM Tool Implementation	19
4.1.1	Development Environment and Dependency Framework	19
4.1.2	k-Tail Algorithm Implementation	20
4.2	Implementation of FSM Testing	21
4.3	User Interface	22

4.4	Visualizing Automata	22
5	Evaluation	25
5.1	Test Cases	25
5.1.1	Unit Testing	25
5.1.2	Code Coverage	25
5.2	Existing Tools Evaluation	26
5.2.1	Tools Used	26
5.2.2	Symbol Extraction Format	27
5.3	Trade-off between Precision and Generalisation based on k	27
5.4	kTail-PSM average execution time	28
6	Case Study	31
6.1	Experimenting kTail-PSM Tool	31
6.1.1	Data Sets	31
6.1.2	Testbed	31
6.2	Inferred Protocol State Machines (TCP/SMTP)	31
6.2.1	TCP Protocol State Machine (PSM)	32
6.2.2	Inferred SMTP Protocol State Machine	33
7	Conclusions	35
7.1	Conclusions	35
7.2	Contributions	36
7.3	Future Work	36
8	Appendix	37
8.1	Customized Export Functionality integrated into Netzob	37
8.2	Lua Script to extract TCP Flag codes	37
8.3	Implementation of k-Tail Equivalent State checking and Merging them	38
8.4	kTail-PSM tool Graphical User Interface	39
8.5	TCP Reference Protocol State Machine	40
8.6	Simplified SMTP Reference Protocol State Machine	40
8.7	Inferred TCP Protocol State Machine Processed Log	41
8.8	Inferred SMTP Protocol State Machine Processed Log	43
8.9	Execution time data sets for ktail-PSM tool	43

Figures

2.1	Automata without k-Tail being applied to it	11
2.2	Minimized automata in Figure 2.1 after k-Tail was applied with $k = 2$	11
3.1	Shows Netzob Architecture[1]	15
3.2	Utility tool showing Network traces being prepared to imported from a PCAP into Netzob	15
3.3	Imported traces in Netzob after applying sequence alignment	16
3.4	Using a Lua Plugin script to create a column of TCP flag codes in Wireshark.	17
3.5	Overview of kTail-PSM tool	18
4.1	A sample directed graph as a state machine generated from Listing 4.1	20
5.1	Shows customized export functionality integrated into Netzob	27
5.2	kTail-PSM average execution time	28
6.1	Inferred TCP client protocol state machine generated using kTail-PSM tool	32
6.2	Inferred SMTP client protocol state machine generated using kTail-PSM tool	34

Chapter 1

Introduction

A protocol is an agreement for exchanging message in a distributed system. It dictates the communication process by defining the syntax and semantics of the messages and message ordering [2]. It is estimated that approximately over 2000 networked applications and about 600 network protocols are in use over the Internet [3][4]. The communication between network applications must conform to agreed protocols in order to exchange messages. However, as many applications use proprietary protocols with no public specification, they may be utilized to carry out malicious activities in the network. For instance some applications such as worms or malicious code may be injected into the network to infect machines on the internet. Consequently understanding of all protocols is increasingly important for network security. Analysing and identifying the details of the proprietary protocols provides insight into their behaviour which can be helpful for traffic monitoring and management purposes. In general, the proprietary protocols contain both a language and a state machine [5]. This information can be captured from both network traces and/or program executions traces. The protocols can then be formally modelled to represent their behaviour and how they are used in communication processes.

Currently, there are existing tools that are aimed at extracting protocol specifications¹ in an automated way. However, most of these tools implement solutions aimed at obtaining message format and vocabulary without their protocol state machines (PSM). Most have also been developed as a proof-of-concept tools and have not been made available to the public to evaluate. To the best of our knowledge, Netzob[1] is the only open-source protocol reverse engineering tool that has been available to the public for evaluation and extension. We evaluated the tool with a number of different network protocol traces and established that the tool works well for extracting protocol specifications for application layer protocols. However, there is no automated way to generate protocol state machine from the extracted symbols. It is a time consuming manual process to generate the protocol state machines (PSM).

Our solution is to develop a tool that will automatically generate PSM from an ordered sequence of protocol symbols. We utilise the Netzob tool to do the necessary processing of the network traces to extract protocol specifications. These specifications are then taken as input into our tool to automatically generate a PSM of using an equivalent state merging algorithm called *k-Tail* algorithm[6]. At this moment, we focused on application-level clear-text PSM such as Simple Mail Transfer Protocol (SMTP RFC² 821) as described Internet Engineering Task Force (IETF). It can be applied to both open and closed protocols to infer their PSMs. In fact, closed protocols are a very interesting target for security purposes

¹In the context of this project, *protocol specifications* refers to an ordered sequence of protocol vocabulary/symbols which help in inferring protocol state machines.

²<https://tools.ietf.org/html/rfc821>

because, as opposed to open protocols, they are not subject to the public scrutiny and testing.

We evaluated our tool with the current specification of the SMTP protocol . We also evaluated the tool with the well known binary protocol, Transmission Control Protocol handshake flags (TCP RFC³ 793) to infer its PSM. We compared the precision of the PSMs against the reference PSMs manually derived from the RFC specifications. The inferred PSMs generated from our tool, captured more detailed state transitions than the reference PSMs. This more detailed PSM is much closer to the real utilisation of the protocol than the original document-based PSM. It can provide valuable information as an unifying state and transition information, which we intend to use in the future for testing and security purposes.

This report continues as follows: Chapter 2 contains the background survey on Protocol Reverse Engineering (PRE) prerequisites, approaches and exploration of automatic PRE tools. Chapter 3 covers the design approaches used for the proposed kTail-PSM tool. Chapter 4 covers how the tool was implemented, based on the design choices provided in chapters 2. Chapter 5 presents the evaluation of the implemented tool called kTail-PSM. Chapter 6 presents the case study conducted using the kTail-PSM tool. Chapter 7 presents the conclusions, contributions and future work. Following these chapters are the appendices.

³<https://tools.ietf.org/html/rfc793>

Chapter 2

Background and Literature Review

This chapter explores concepts relevant to the project, conducting a background survey on the PRE approach and PRE tools that are available.

2.1 Protocol Reverse Engineering (PRE) Prerequisites

PRE is defined by UC Berkley's BitBlaze¹ project as *the process of extracting the structures, attributes, and data from a network protocol implementation without access to its specification*. Usually, it is a manual task which is time-consuming and error-prone. PRE can be classified in terms of the two data input sources from which protocol specifications can be extracted. The network trace and the program trace from an executable program. The input requirements for PRE are discussed next.

2.1.1 Network Trace-based Input

When using the network trace-based input, network traffic of a specific application such as SMTP, HTTP, DNS, BitTorrent, etc. is collected using packet capture tools such as *wireshark*² or *tcpdump*³[7]. For instance, Zhang et al.[8], uses wireshark to capture the communication traffics between target protocol implementation and a client. The session data is logged into a *pcap* file. The *pcap* is a packet interchange file format that allows captured network packets to be stored for analysis with other packet analysing tools. This logged network trace is then analysed to extract the protocol specifications.

2.1.2 Program-based Input

Another input source used in the process of analysing and identifying undocumented protocols is executable programs. As the target is a proprietary program, there is no way to get access to the protocol specification in any form. Hence, in this method it deals with the binary code extracted from executable program using other program-debugging tools such as *OllyDbg*⁴. For instance, Medegan [9] applied *OllyDbg* debugging tool and other plugins to debug Skype executable in order to reverse engineer its protocol.

¹<http://bitblaze.cs.berkeley.edu/protocol.html>

²<https://www.wireshark.org/>

³http://www.tcpdump.org/tcpdump_man.html

⁴<http://www.ollydbg.de/>

2.1.3 Target Protocol Encoding

One of the focuses of the protocol PRE is mining the protocol communication data from the two data input sources described in sections 2.1.1 and 2.1.2. The communication protocols are categorized based their nature of encoding.

- **Text-based Protocol** — A text-based protocol refers protocol messages that may be human readable. In other words, the protocol content is presented in plain text format. For example, protocols such as IRC, SMTP or HTTP etc. are text-based protocols.
- **Binary Protocol** — A binary protocol is a protocol whose content presentation is for machine readable format as opposed to a text-based protocol. For example, protocols such as DNS, SMB, TCP, IP etc. are binary protocols.

2.2 Protocol Specification

Protocol specifications consist of three dependent components [10]: (1) protocol syntax, (2) protocol semantics including which data values are valid in defined protocol fields , and (3) the protocol Finite State Machine (FSM)⁵ which will be discussed in Section 2.2.3. In the context of this project, we focus on the on (3) — i.e. inferring FSMs. In order to infer an FSM, it requires the extraction of vocabulary (symbols) and grammar (valid sequences of messages) from protocol traces.

2.2.1 Protocol Vocabulary

Protocol vocabulary [7] comprises a set of symbols where each symbol represents an abstraction of similar messages. This similarity property refers to messages having the same role from a protocol perspective. For instance, the set of TCP ACK messages can be abstracted to the same symbol.

2.2.2 Protocol Grammar

From the perspective of network protocols, protocol grammar describes the sequence of messages that are exchanged in a valid communication between two actors. For instance, in the ICMP protocol, its grammar includes a rule which states that an ICMP ECHO REPLY TYPE 8 always follow an ICMP ECHO REQUEST TYPE 8 [7]. Another example is the ordered messages sent between two actors in a TCP sessions. In this example, the ordered set {SYN,SYN+ACK,ACK} is valid while {SYN+ACK,SYN,ACK} is invalid as SYN should be the first message sent from the initiating actor and not SYN+ACK. By inferring the protocol grammar from the network trace, we are able to infer the protocol state machine.

2.2.3 Finite State Machines (FSM)

An FSM is also known as deterministic finite state automaton (DFA) or finite state automata (FSA). From the perspective of network protocol, it is referred to as protocol state machine (PSM). These terms will be used interchangeably in the following sections of this report— FSM is defined by a set of states, exactly one of which is the *initial state*, some subset of which are the *accepting states*, a set of input tokens called the *alphabet*, and a transition function that, given the next input token and the current state, determines the next state [11].

⁵In the context of this project, protocol PSM and protocol FSM mean the same thing

The protocol FSM defines the order, and state in which messages are exchanged. The process of inferring protocol FSM from a trace requires as input the protocol grammar described in Section 2.2.2. There are various modelling techniques that are used to create protocol FSMs. A commonly used technique is Mealy machine. For example, in a Mealy machine, the output is decided from the current state and the outgoing transition. This can be illustrated as: $\{state, input\} \rightarrow output$.

2.3 Protocol Specification Extraction Methods

Extracting protocol specifications from network traces or binary executables is a challenging task as these sources provide incomplete information. To utilise such limited information, a number of different PRE approaches have been introduced. This section discusses the automatic PRE techniques proposed by various studies undertaken to reverse engineer proprietary protocols. These techniques utilise a number of automatic PRE tools which will be discussed in section 2.4.

2.3.1 Grammatical Inference

Grammatical inference (GI) (also known as grammar induction, or grammar learning) deals with idealized learning procedures for acquiring grammars on the basis of exposure to evidence about languages [12]. This technique has been introduced in the context of PRE to extract protocol specifications automatically. GI approach has been applied to extracting protocol specifications from both network-based traces and program-based inputs. Moreover, GI approaches have been applied to both text-based and binary protocols.

Ming-Ming and Shun-Zheng (2011), introduced a grammatical inference algorithm to model protocol state from which they derive regular grammar i.e, the FSM and further applied equivalent state merging algorithm to generate a generalized and minimal automata [2].

Xiao and Yu (2010) and Zhang et al. (2012) , introduce grammatical inference to model protocol specification as FSM from the extracted network trace. Their work involves mining protocol state machines by interactive grammar inference technique. That is using grammatical induction as a learning process to generate queries to the protocol implementation which could help infer the automata [13][8].

2.3.2 Statistical Analysis

Statistical analysis involves the application of statistical models to extract protocol specifications. Statistical analysis methods have been applied to the PRE context to automatically extract proprietary protocol specifications for both binary and text-based protocols. To the best of our knowledge, this approach has only been applied to network-based traces.

Meng, Liu, Zhang, Li and Yue (2014), proposed a statistical analysis approach to infer protocol state machine from binary protocols solely based on real-world network traffic of a specific application [14]. In this study, the authors presented a methodology to align corresponding fields and extract state fields from binary protocol communication traces. Based on these state fields, the system can construct the protocol state model.

Wang,Y., et al. (2011), proposed a system that can automatically infer protocol state machine from real-world network traces [15]. Based on the statistical analysis on the protocol message formats, the system infers protocol state machines without prior knowledge of protocol specifications. The output of the system is a state transition model called a proba-

bilistic protocol state machine (P-PSM). This representation is a probabilistic generalization of a protocol state machine.

In another study by Wang, Li, Meng, Zhao, Zhang and Guo (2011), they proposed a system that automatically extracted the binary protocol message format of an application from real-world network trace [16]. The core concept of the system is to find the same format information in a specific protocol. The output of the system is a transition probability model built based on the statistical nature of protocol format.

2.3.3 Keyword Identification

Keywords are predefined constants in protocol messages. Keywords can appear in binary, text, and mixed protocols and can be strings or numbers. For example, the *GET*, *HTTP*, *User-Agent*, and *Host* strings are all keywords [17].

Cui, W., et al (2007), achieve keyword identification by tokenization and initially clustering the messages in the network traces [18]. This process is applied on raw packets and helps in identifying field boundaries in a message and giving the first order structure to unlabelled messages. The packets are first re-assembled into messages and then breakup a message into a sequence of tokens which is an approximation to a sequence of fields.

Wang, Zhang, Wu and Su (2013), proposed a framework which can infer unknown protocol specifications and the FSM based on keyword identification [5]. In this framework, network traces of a specific protocols are collected and the packets in the traces are assembled into messages. These messages are then tokenized and a threshold is adopted to filter out tokens. The threshold determines that the keywords with low frequencies are useless. By varying the threshold, the framework was able to get the optimal tokens set. These optimal tokens are the keywords for the text protocol. From these keywords, the framework creates a finite state machine for protocol language inference (L-FSM). A similar study was conducted by Luo and Yu (2013) to infer protocol state machines from network traces [19]. In their study, the traces (tcp dump files) are parsed and application layer sessions are reconstructed based on five-tuple analysis which they define. From this, protocol keywords are extracted. That is, the frequent strings are extracted and variance analysis is applied to mine protocol keywords. Next, message format is extracted based on the keyword series set and infers fields of message format based on keywords series. Messages in each session is labelled according to their formats. Then protocol state machines are inferred by searching for frequent subsequence of labels.

2.3.4 Static and Dynamic Analysis

These approaches always take an executable program to determine information about the program with respect to its protocol specifications. Program analyses can be divided into two categories according to when they are analysed [20]: (1) Static Analysis — This approach involves analysing a programs source code or machine code without running it. (2) Dynamic Analysis — This approach involves analysing a client program as it executes. In other words, it has the ability to monitor code as it executes. One of the attractive features of Dynamic analysis is that it allows us to reason about actual executions, and thus can perform precise analysis based upon run-time information. Dynamic analysis has been a widely used approach in the extraction of protocol specifications [21][22][23][24][25][26][27].

Wang, Y., et al. (2013), combines both static binary analysis and dynamic binary analysis approaches to effectively infer the message format of a protocol of a target program [28]. Since a single protocol message contains a large number of fields (such as non-static fields

with variable sizes and inter-dependency of fields etc), combining the two approaches provides the capability of identifying field attributes more effectively than only one approach.

On top of dynamic analysis, Wang, Z. (2009), introduced an approach called *Dynamic Taint Analysis* [29]. This approach was first proposed by Newsome, J., and Song, D., (2005) but in different problem domain [30]. However, since then it has been adopted to the context of protocol reverse engineering domain. He, Y., et al. (2009), adopt both *dynamic analysis* and *dynamic tainted analysis* approaches to extract protocol format from data flow information revealed by application while processing the protocol data [31].

2.3.5 Other Approaches

There are three other approaches that have been introduced to extract protocol specifications. Wang, Y., et al. (2012), proposed a *semantics-aware* approach to discover the latent relationship among n -grams by first grouping protocol messages with the same semantics and then inferring message formats by keyword based clustering and cluster sequence alignment [32]. In this approach, it takes a network trace as an input and outputs the inferred protocol message format. It is applicable to both binary and text-based protocols.

Caballero, J., et al. (2007), introduces an approach called dynamic data flow analysis to reverse engineer network message formats [27].

Whalen, S., et al. (2010), introduces a HMM-based (Hidden Markov Model) approach for inferring the state machine [33]. The Markov property states that the conditional probability distribution of a system at the next time period depends only on the current state of the system, i.e., not depending on the state of the system at previous time periods.

2.4 Classification of Automatic PRE Tools

To explore the existing automatic PRE tools, we categorize them into two categories: (1) tools that focus on extracting protocol symbols and message format (protocol syntax), and (2) tools that focus on determining the protocol FSM. Many FSM PRE tools require protocol symbols and the syntax information. However, vice-versa is also true since *Syntax/Message Format* PRE tools require protocol FSM information. In this project, we choose to present message format PRE tools first based on the observation that historical research targeted message format PRE methods prior to FSM PRE methods. These two categories are discussed next

2.4.1 Protocol Message Format/Syntax Extraction Tools

Discoverer[18] — Cui, W., et al. (2007), proposed a tool called Discoverer which implemented the keyword identification and common protocol idiom approaches to automatically reverse engineer the protocol message format of an application from its network trace.

PolyGlott[27] — Caballero, J., et al. (2007), introduced an automatic method of extracting protocol message formats by applying dynamic binary analysis approach. They implemented a tool called Polyglot, which extracts protocol information by observing the execution of a program while it processes execution traces to detect the fields which compose a message. It can infer some field semantics, such as detecting keyword fields and direction fields (which can be either length or pointer fields). Polyglot takes both input types, network-based traces and program-based.

AutoFormat[34] — Lin, Z., Jiang, X., et al. (2008), proposed another tool call AutoFormat, improved from Polyglot by revealing the inherently non-flat, hierarchical structures

of protocol messages using dynamic analysis approach. It takes both network-based and program-based traces.

Tupni[23] — Cui, W., et al. (2008), proposed a tool called Tupni. Given one or more inputs of the unknown format and a program that can access these inputs, Tupni can reverse engineer an protocol with rich set of information. Tupni can identify arbitrary record sequences by analysing loops in a program, using the fact that a program usually processes an unbounded record sequence in a loop. This automatic protocol reverse-engineering tool uses dynamic analysis approach to analyse data from both input types, network-based traces and program-based.

Prospex[25] — Comparetti, P.M., et al.(2009), proposed a tool called Prospex to extract message format specifications. The system analyzes both binary execution traces combined with network traffic. The tool uses dynamic taint binary analysis approach to discover how an application processes its incoming data. During the session analysis phase, it applies message format inference technique to extract protocol message format for a single message.

Dispatcher[24] — Caballero, J., and Song, D. (2009), proposed an automatic protocol reverse engineering tool called Dispatcher based on dynamic program analysis approach which leverages the ability of program that implements the protocol to extract the protocol message format. It is purposely to reverse engineer undocumented command and control (C&C) protocol of MegaD spam botnet.

DynamoRIO[31] — He, Y., et al. (2009), proposed an PRE approach to determine unknown protocols based on data flow analysis using a framework called DynamoRIO, which is a fully implemented run-time code manipulation system supporting dynamic taint analysis and allows code transformation on any part of a program while it is executing. Dynamic taint analysis approach is applied to the data flow analysis process to reverse engineer the protocol extraction. This is a program-based approach to extract protocol message format which is not applicable to network traces.

ReFormat[29] — Wang, Z., et al. (2009), proposed a program-based tool called ReFormat that reveals how a program parses and processes a message. ReFormat handles encrypted messages by providing an effective scheme to discern the protocol processing phase from the message decryption phase and then pinpoint the run-time memory buffers that contain the decrypted message. It achieves this by applying dynamic taint analysis approach and further relies on another general technique, i.e., data lifetime analysis, to locate the decrypted memory buffers.

Biprominer[16] and ProDecoder[35] — Yipeng, W., et al. (2011), proposed Biprominer and ProDecoder, that used statistical methods to find keywords and probable keyword sequences. Biprominer was intended for automatically extracting binary protocol message formats of an application from its real-world network trace. It operates in three phases: (1) uses statistical analysis to identify relevant patterns for specific pattern lengths and identified them as keywords, (2) messages are defined with distinguishing keywords, and (3) transitions between keywords are calculated to find probable message sequences of keywords. Biprominer uses variable length pattern recognition to find distinguishing protocol keywords. ProDecoder uses the same first and second phase, but then differs by using a clustering algorithm followed by the Needleman-Wunsch[36] algorithm for text alignment. To learn distinguishing keywords in phase 1, Biprominer and ProDecoder find binary patterns of arbitrary length, called n -grams, where n denotes the number of bytes in the pattern. ProDecoder targets text-based as well as binary protocols.

NetProtocolFinder [37] — Ying, W, et al. (2013), proposed an automatic protocol message format extraction tool called NetProtocolFinder, which is designed and implemented to analyse documented and undocumented protocol message format automatically. It is a program-based tool that combines dynamic binary analysis and static binary analysis ap-

proaches to identify the field attributes more effectively. In the paper, a number of text-based protocols were tested with the application and the results showed that the combination of two analysis techniques showed its effectiveness in inferring the message format.

2.4.2 Protocol State Machine Modelling Tools

In this section, we discuss PRE tools that infer the protocol state machine model. A protocol state machine characterizes all possible legitimate sequence of messages. This builds up from sub-section 2.4.1 because it captures the relationship between the protocol message format and the protocol state machine. A number of automatic PRE tools have been proposed recently for generating an output representation in the form of state machines.

ScriptGen[38] — Leita, C., et al. (2005), proposed a tool called ScriptGen, to alleviate problems existing in Honeyd[39]. It was designed to generate honeypot scripts. It includes features that both address the problem of vocabulary and the grammatical inference. The tool extracts messages exchanged between a client and server(honeyd) from *tcpdump* file, by adding all observed messages one by one to build a state machine. It uses the protocol automaton to identify similar messages and passively builds a finite state machine by replaying the various sessions provided in the traces.

PEXT[21] — Shevertalov, M., et al. (2007), proposed a dynamic analysis tool called PEXT to utilise network traces to infer an approximate state machine by analysing a collection of packets captured from an application at run-time. An approximate state machine is inferred by clustering messages of the same type, based on distance metric and by analysing the similarities between different sequences of types present observed the traces. This approach is useful to evidence patterns of sequences of messages that arise from using specific protocol features. However, it cannot derive the message formats, creating a semantic gap between the final automaton and the observed data.

ReverX [40] — Antunes, J., et al. (2011), proposed a tool called ReverX for automatically inferring the language and state machine of a given protocol by applying the GI approach. It constructs two automata, one for language and the other for the protocol state machine from the sequences of messages and protocol sessions that were observed in the network traces, then generalizes and reduces them in order to create a concise specification. The methodology was implemented in a tool called ReverX which takes only text-based protocols.

Veritas[15] — Yipeng, W., et al. (2011), proposed Veritas, a system that automatically infers protocol state machine from real-world network traces. It is based on the statistical analysis on the protocol formats and relies on technique to cluster equivalent messages.

Netzob[1] — Bossert, G., et al. (2012), introduced Netzob, a tool that allows dynamic analysis of proprietary protocols. It leverages passive and active algorithms on observed communications to build a model. This model can afterwards be used to measure the conformity of an implementation against its provided specification. It learns message formats and state machines of protocols, and thus can identify deviations with the documentation. Netzob handles both text-based and binary protocols as well as other protocols such as variable fields protocols (like ASN.1 based formats). There are a few limitations found in the tool. One of the is that the state machine creation requires the states are created manually. For a novice user, this feature may be puzzling as the user might expect the system to generate the protocol state machine automatically. In other words, the state machine is generated based on specifying the states and transitions using a manual constructor. To the best of our knowledge, Netzob is the only tool that is open-source and available to the public on git⁶ repository.

⁶Netzob source code: <https://github.com/netzob/netzob>

AutoReEngine[19] — Luo, J. Z., and Yu, S. (2013), proposed a novel approach for position-based reverse engineering for network protocols. It is purposely to extract protocol keywords from network traces based on their support rates and variance of positions, reconstruct message formats and infer a protocol state machine. The approach was implemented by modifying the *Apriori* algorithm to exchange frequent strings from application layer payload. Then, position-based features are considered to select protocol keywords from frequent string set via variance analysis. Based on the protocol keywords, keyword series set is found and fields of message formats are inferred. Finally, the most frequent communication patterns are inferred as the protocol state machine. The approach was implemented by developing a tool called AutoReEngine which was evaluated with real-world traffic, covering both text-based and binary protocols.

2.4.3 Availability of automatic PRE Tools

Protocol Reverse Engineering research is quite active in academia. Most of the proposed automatic PRE tools discussed earlier are not available to the industry domain. Most of the tools have been developed as a proof-of-concept tools. There are also other packet analysing tools which are available such as wireshark and tcpdump. However, such tools are only useful for manual analysis approaches. Currently, Netzob is the only active project in the open-source community. Also, since it comes with a framework of APIs, it is easy to work with and allows to integrate customized functionality easily to it.

In this project, we employed Netzob as our primary tool to extract protocol specifications. In particular, we used the tool to extract protocol symbols from network traces and then we process them in our proposed kTail-PSM tool to infer the protocol state machine.

2.5 k-Tail Algorithm Overview

The intuition behind k-tail[6] algorithm is that if two states have identical, k-long sequences of observed events following them, then those states are assumed to represent the same state. Therefore, to infer a concise model, k-Tail algorithm merges states that it considers to represent the same state. The process stops once all points deemed equivalent are merged. The parameter k determines the size and generality of the inferred model—a smaller k leads to more merges and produces more compact (and more general) models, while a greater k restricts state equivalence[41]. Listing 2.1 shows the k-tail algorithm.

```

1 Input: Log L, int k
2 let M = initial FSM model of traces in L
3 let merged = true
4 while (merged):
5     merged = false
6     foreach (States s1, s2 in M):
7         if (s1, s2 are k-equivalent):
8             M.merge(s1, s2)
9             merged = true
10
11 Output: M

```

Listing 2.1: The k-Tail algorithm[41]

Let's consider the following sequence of symbols: $\Sigma = \langle a, b, c, d, a, b, c \rangle$ and its canonical automata is represented as shown in Figure 2.1. We discuss the generation of states and transitions shortly in Section 3.3.5.

In order to determine how the algorithm works, let's take $k = 2$. The breakup of the canonical automata is given in Table 2.5.

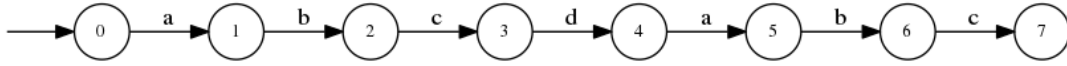


Figure 2.1: Automata without k-Tail being applied to it

state	k-tail ($k = 2$)
0	a,b
1	b,c
2	c,d
3	d,a
4	a,b
5	b,c

Table 2.1: Breakup of canonical automata in Figure 2.1 given $k = 2$

When we apply the k-Tail algorithm to this sequence with $k = 2$, it can be seen that states 0 and 4 have 2-long sequences of symbols $\{a, b\}$ and also states 1 and 5 have 2-long sequences symbols $\{b, c\}$ following them respectively. Therefore, those states are assumed to represent the same state. At this stage, the last k -long tail of the initial FSM is dropped based on the value of k . In this case the states 6 and 7 are dropped from the FSM given in Figure 2.1 since $k = 2$. Hence, after merging the equivalent states, we obtain a minimized automata as given in Figure 2.2.

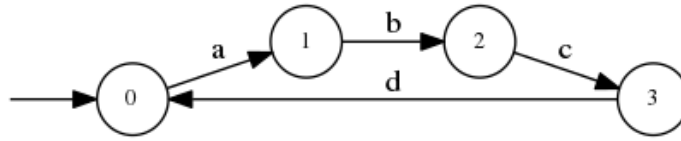


Figure 2.2: Minimized automata in Figure 2.1 after k-Tail was applied with $k = 2$

The detailed explanation of merging two equivalent states is discussed in the implementation chapter in Section 4.1.2.

2.6 Using FSM for validation and Parsing Tokens

FSMs are useful for testing input tokens for determining acceptable behaviour of an implemented protocol. This requires a deterministic FSM that recognises the language of all valid messages in that protocol. For instance, a security policy might be concerned with access control, and restrict what operations can perform on objects. The access to the restricted objects can be validated through acceptance testing based the reference security policy model. This means the access to the restricted object can be only accepted if an only if the access right to the object is in an accepting state when the validation reaches the end and passes all validation tests. Any access right that is not accepted is rejected. An example of the application of this technique in the security context is given by Schneider[42]. In his work, he introduces an automata-based formalism for specifying security policies that are enforceable with mechanisms that work by monitoring system execution. Another example of the application of this technique is given by Graham D.,W et al. [11]. In their work, they applied this technique by implementing an FSM-based parsers for parsing protocol specific files.

In this project, we implemented the feature to check for the acceptance or rejection of input symbols given a reference FSM. The implementation of this functionality is discussed in Section 4.2

Chapter 3

Design

This chapter explores the engineering constraints and challenges of integrating additional functionality with Netzob to automatically generate state machine from a set of extracted protocol specifications (symbols) from network traces.

3.1 Engineering Constraints

After a thorough revision of techniques and available tools, we chose to use Netzob to analyse protocol specifications and to extract protocol vocabulary. Netzob[1] is discussed in Section 2.4.2 and is an open-source tool is developed in Python and is distributed under the GPLv3 license.

It comes with a graphical user interface that allows users to analyse and infer protocol state machines in a semi-automated way for communication protocol data imported from network traces. Netzob version¹ 0.4.1 (stable) was used for this project. In the process of evaluating the tool with different protocols, it was observed that the tool has two limitations:

1. The export functionality was not working. Hence, we had to add export functionality in order to export the extracted symbols into a comma-delimited file format.
2. Consequently, it seems to filter out some information fields from the input traces. For instance, when importing TCP packet streams, none of the information about the TCP flags were identified in Netzob. We will discuss this in more detail in section 6.2. Netzob is suitable mostly for reverse engineering application-layer protocol payloads.
3. After identifying the limitations in Netzob, we looked at Wireshark² tool as an alternative choice. Wireshark is a packet sniffing tool that can be used to analyse traffic. The tool allows the analysis of network traffic in detail. For instance, we used the Wireshark tool to extract the contextual information we needed to identify the TCP handshake process, in particular to extract TCP handshake flags. However, wireshark is not able to infer protocol state machines.

3.2 Design Goals

In this section, we identify the design goals for protocol specification extraction and inferring a state machine from the extracted protocol specifications. The goals were identified

¹<https://www.netzob.org/>

²<http://www.wireshark.com>

as a result of the engineering contrants identified in Section 3.1.

1. Extract the protocol symbols from network traces and export the specifications into a comma-delimited file while preserving the order.
2. Introduce a tool to automatically generate FSM for protocol symbols extracted in goal 1
3. Parsing protocol symbols against reference FSM to determine its behavior.
4. Expand the GUI-based user interface enable users to be able to import a sequence of extracted protocol symbols and generate an automata automatically.

3.3 Design Approach

In order to achieve these design goals, we proposed a simple FSM generation tool which can be integrated into Netzob. Utilising Netzob APIs, we are able to extract protocol symbols which are then fed as input to the proposed tool. The k-Tail[6] inference algorithm is selected for modelling the finite state machine for the sequence of input symbols. The algorithm is discussed in section 2.5. Wireshark compliments Netzob by analysing and extracting specific fields that are related to protocol state machines (PSM) especially for binary protocols. It was also used to cross-check Hex codes for symbols extracted in Netzob.

In the next two sub-subsections, we take a detailed look at the Netzob and Wireshark tools and the additional functionalities that are integrated into them respectively.

3.3.1 Using Netzob

- **Netzob Features and Functionality**

Netzob implements clustering and sequence alignment algorithms to process the raw data from the network traces. There are two approaches used that result in extracting protocol symbols:

1. GUI-based functionality
2. Netzob APIs.

For the purpose of this project and to get some sense of user experience with the Netzob tool, we utilized the GUI-based approach to extract protocol symbols. Figure 3.1 shows an overview of the process involved in extracting the protocol symbols.

The Netzob[1] architecture, comprises of four core components connected through well documented APIs. These are:

- Import Module: This module provides functionality to import data into Netzob. It provides the capability of capturing data in a variety of context such as from inter-process communications (IPC) as well as capturing from wired network traffic. Additionally, it support input formats such as network flows, PCAP files, structured files and IPC (pipes, socket, shared memory).
- Protocol Interface Module: This module provides the vocabulary and grammer extraction functionality of Netzob. That is, the vocabulary and grammar inference methods constitute the core of Netzob.

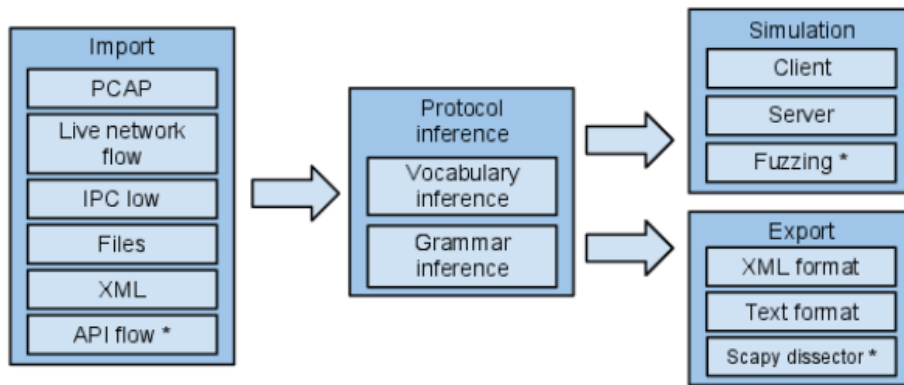


Figure 3.1: Shows Netzob Architecture[1]

- Simulation Module: As one of the main goals of Netzob is to generate realistic network traffic from undocumented protocols, this module provides the simulation functionality.
- Export Module: This module permits to export an inferred model of a protocol in formats that are understandable by third party software or by a human. Current work focuses on export format compatible with main traffic dissectors (Wireshark and Scapy) and fuzzers (Peach and Sulley).

In this project, we utilised only the *Import Module* and *Protocol Interface Module* to extract the protocol symbols and developed a new module to accept the protocol symbols and generate FSM. The new module is called kTail-PSM. The input source we chose for this project is basically a *PCAP* file format containing a set of network traces for a given protocol.

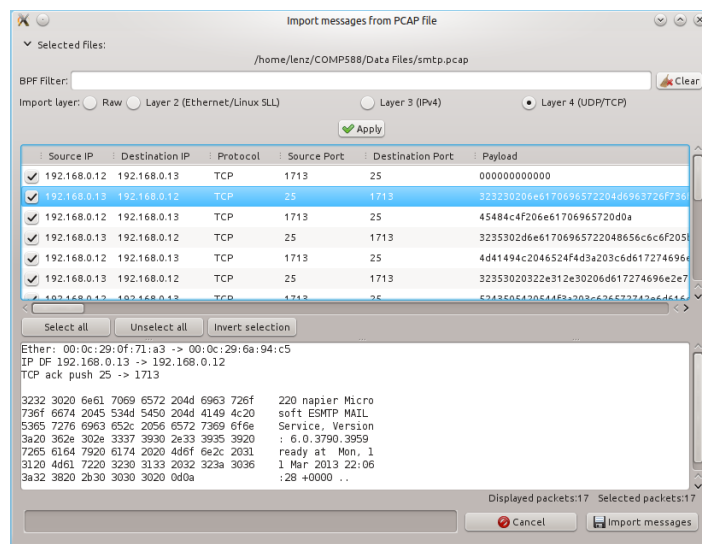


Figure 3.2: Utility tool showing Network traces being prepared to imported from a PCAP into Netzob

Figure 3.2 shows PCAP file import details of a sample of SMTP protocol in Netzob. For the convenience of our project, we chose to work with *PCAP* files as they are easily imported and processed in Netzob. Once the set of traces are imported into Netzob,

we then apply the clustering and sequence alignment algorithms to extract the protocol symbols. To discover a symbol, Netzob supports different partition approaches. One of them is the Needleman-Wunsh algorithm[36] that leverages sequence alignment processes. The alignment process applied on the messages using the Sequence Alignment functionality as shown in Figure 3.3. The objective is to identify common messages, to regroup them in dedicated symbols, and to obtain a field partitioning of each identified symbol. This gives a relevant first approximation of the overall message format.

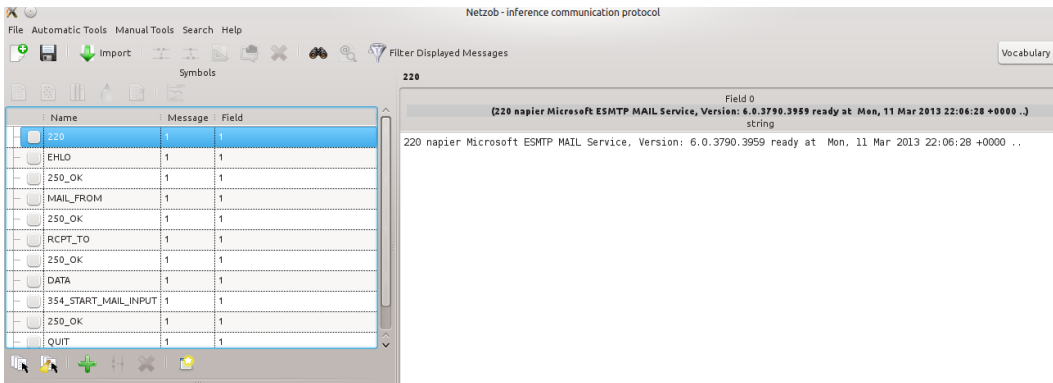


Figure 3.3: Imported traces in Netzob after applying sequence alignment

The next step is to export the extracted protocol symbols from Netzob to a file as an ordered sequence of protocol symbols ready for importing into the kTail-PSM tool to infer the state machine. Since, Netzob's export functionalities do not allow us to customize processed data to suit our requirement, we integrated an additional functionality to extract only the required data as discussed next.

- **Customised Features integrated into Netzob**

The proposed *kTail-PSM* tool takes as input an ordered set of comma-delimited protocol symbols. To meet this requirement, we modified only the *Export Module*. More specifically, the following module was modified:

- Module name: *src.netzob.UI.Export.Controllers.RawExportController*
- File name: *RawExportController.py*

The code fragment is attached in Appendix 8.1. The modified version of the Netzob tool is available on git³.

3.3.2 Using Wireshark

Wireshark compliments Netzob in analysing network traces, as Netzob is not able to handle specific information about binary protocols. For instance, we experimented with Netzob to extract TCP flags from a set of network traces imported into Netzob. However, such information was not easily identifiable in Netzob. Thus, we resort to Wireshark to extract the TCP flags. On the other hand, the tool worked well with the application level protocol payloads such as SMTP payload.

- **Wireshark Features**

Wireshark⁴ is an open-source network packet analyzer tool to capture network packets

³<https://github.com/lnerit/Netzob-0.4.1>

⁴<http://wireshark.com/wireshark-network-packet-analyzer/>

and tries to display the packet data as detailed as possible. It comes with a number of features. In this project we made use of the following features:

- Import/Export reports to plain text, CSV, PostScript, PCAP and XML.
- Packet Filtering system.
- Plugin Support feature

Wireshark captures live network traffic which can then be exported into a file format supported in Netzob. Another, interesting feature about Wireshark is that it comes with plugin support. In our case, there is no straight-forward way to extract TCP flags in Wireshark while maintaining the order in which they are transmitted. As a result, we utilised the plugin capability of Wireshark, which is discussed next.

• Utilising Wireshark Plugin Feature

Wireshark allows creating a plugin to extract details of specific fields in a protocol. In this project, we used Lua script as a plugin to Wireshark to create a field/column for TCP flag codes in the trace display window as shown in Figure 3.4. In order to execute the Lua script, the script must be placed into the plugin file in Wireshark. For instance, in this project, the script named *tcp-flags-dissector.lua* was placed in the following directory: *~/wireshark/plugins/*. On restarting Wireshark, the script automatically takes effect.

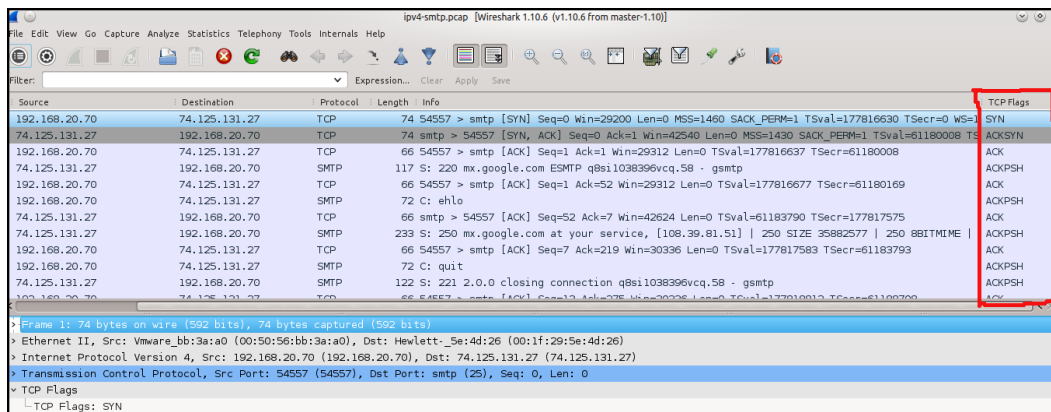


Figure 3.4: Using a Lua Plugin script to create a column of TCP flag codes in Wireshark.

The customized Lua script to extract the TCP flag codes is appended in appendix 8.2. Thanks to Didier Stevens for the Lua scripting tutorial⁵ in Wireshark.

3.3.3 k-Tail Protocol State Machine(PSM) Tool

This section covers the main design for the proposed kTail-PSM tool. As discussed earlier, the tool takes a set of ordered protocol symbols as input and applies the k-Tail[6] algorithm to merge equivalent states in the sequence. The k-Tail algorithm is discussed earlier in Section 2.5. After the merging of equivalent states, the tool generates a PSM. Figure 3.5 presents a high level overview of the kTail-PSM tool.

⁵<http://blog.didierstevens.com/2014/04/28/tcp-flags-for-wireshark/>

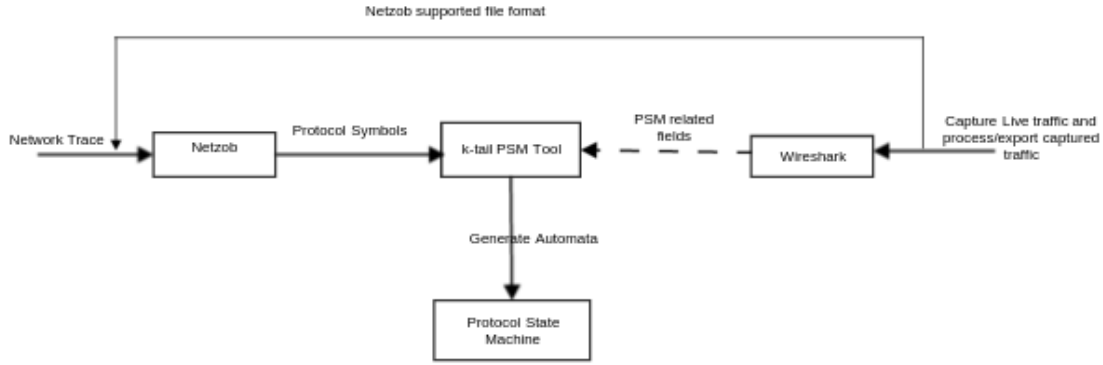


Figure 3.5: Overview of kTail-PSM tool

The input to the tool is an ordered sequence of comma-delimited protocol symbols which are extracted from Netzob. In Figure 3.5, the dotted line from Wireshark indicates that specific PSM related fields in a protocol such as TCP flag codes, can be extracted and fed into kTail-PSM tool to generate an inferred protocol state machine.

3.3.4 Finite State Automata (FSA) Representation of Protocol Symbols

By describing systems in a formal, mathematical way, it allows us to reason about systems both in general as well as specific instances. Formal notations also help to eliminate ambiguity in the implementation of the system design. To make the concept clear, we provide a brief introduction to the formal notations of FSAs.

- **Definition:** An FSA is defined as a five-tuple $\mathcal{A} = \langle \Sigma, \mathcal{Q}, q_0, \delta, \mathcal{F} \rangle$ where:

- Σ is the finite input alphabet
- \mathcal{Q} is the finite, non-empty set of states
- q_0 is the initial state $q_0 \in \mathcal{Q}$
- δ is the transition relation such that $\delta : \mathcal{Q} \times \Sigma \rightarrow \mathcal{Q}$

Note that here for simplicity, we assume that the automaton is deterministic, but the definition can be generalised to non-deterministic and probabilistic automata.

- \mathcal{F} is accepting states: $\mathcal{F} \in \mathcal{Q}$

In this context, the alphabet of the automaton is the set of symbols extracted from Netzob. The application of this definition will be discussed in Section 4.2

3.3.5 Defining States and Transitions from a given Sequence of Symbols

In this project, the input of kTail-PSM tool is a log — an ordered sequence of protocol symbols. For a given set of protocol symbols, we use the following notation to define the states and the transitions from one state to another:

Given a sequence of symbols: $\Sigma = \langle I_0, I_1, I_2, \dots, I_n \rangle$,

we derive the states and their transitions as: $S_0 \xrightarrow{I_0} S_1, S_1 \xrightarrow{I_1} S_2, \dots, S_n \xrightarrow{I_n} S_{n+1}$

The symbols represent the transition labels from their respective associated state to another state. We will use this notation to derive states and transitions from an ordered sequence of protocol symbols in Chapters 4 and 6

Chapter 4

Implementation

4.1 kTail-PSM Tool Implementation

In this Section, we describe the implementation of the kTail-PSM tool.

4.1.1 Development Environment and Dependency Framework

The kTail-PSM tool was developed in Python on a 64-bit version of Ubuntu 14.04. Below is the list of tools and libraries that were used in the implementation process.

- Tools: Eclipse v4.3.2 with PyDev IDE plugin
- Graphviz library for creating directed graphs to represent state machines. Below is a sample directed graph produced from Graphviz library. Listing 4.1 shows the DOT code fragment and Figure 4.1 shows the directed graph output.

```
1          digraph {
2              graph [ accepting_states=1,
3                      rankdir=LR,
4                      ratio=auto ,
5                      size=10,
6              ];
7              node [ fillcolor=white ,
8                      fontsize=10,
9                      height=0.05 ,
10                     label="\N",
11                     shape=circle
12             ];
13             null    [ label=" ",
14                     shape=plaintext ];
15             null -> 0;
16             0 -> 1  [ label=a ];
17             1 -> 2  [ label=b ];
18             2 -> 0  [ label=c ];
19         }
```

Listing 4.1: Code segment to generate a directed graph using DOT language in Graphviz library

The implementation of the *k-Tail*[6] algorithm included an open-source publicly available Python-FSM¹ modules. The Python modules are available to the public on git repository under BSD license. They were included in our implementation to provide the framework

¹fsm python modules: <https://github.com/oozie/python-fsm>

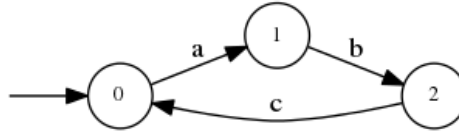


Figure 4.1: A sample directed graph as a state machine generated from Listing 4.1

to generate the inferred state machines after the ordered set of protocol symbols have been processed.

4.1.2 k-Tail Algorithm Implementation

In the implementation of the k-Tail algorithm, we assumed that an input is a single line of comma-delimited protocol symbols for a particular protocol under analysis. Basically the k-Tail algorithm involves two main steps. It looks for equivalent state relations and then merges those equivalent states to get an inferred FSM.

Equivalent States

Given an ordered sequence of protocol symbols, we create an FSM that describes the states and transitions between these states. Based on this, we define an equivalent FSM based on the value of k .

The function in Listing 4.1 determines relations between states in a constructed automata of a given set of protocol symbols. It takes in two states as parameters: s_1 and s_2 and compares them based on their k-Tail equivalence. If k-long sequences of observed transitions following them have identical sequences, then those states likely represent the same state. Hence the function returns value *True* or *False* otherwise.

```

1 def check_equivalence(s1, s2):
2     flag=None
3     if s1==s2:
4         flag=True
5     else:
6         flag=False
7     return flag

```

Listing 4.2: Function to check for equivalent relations between states

Referring to the example in Figure 2.1, when passing states as parameter to $s_1 = 1$, and $s_2 = 4$ will return *True*, as will states 1 and 5 since they have k-Tail equivalence given $k = 2$. The calling function is given in Appendix 8.3.

Merging Equivalent States

Once the equivalent relations are defined, the algorithm then uses this relation to reduce the FSM, meaning the equivalent states are merged. It keeps finding equivalent states and reduce the FSM as necessary until there are no more equivalent relations and the algorithm terminates.

We utilized List data structures in Python to implement the merging of equivalent states. Let's refer to the example in Section 3.3.4. Given the sequence of symbols $\Sigma = \langle a, b, c, d, a, b, c \rangle$ and $k = 2$. Firstly, we define a list data structure to hold the symbols:

```

1 sequence=['a', 'b', 'c', 'd', 'a', 'b', 'c']

```

Next, we derive the states associated with each symbol respectively by taking the index of each symbol. We based on our denition in Section 3.3.5 to define the states and the transitions. In our implementation in Python, we used the *range()* function. This function generates numbers up to but not including the upper bound number. Hence, we added one to it to maintain the accuracy of the calculation when total length of sequence is subtracted by *k*.

```

1 kTails.state=[]
2 for x in range(0,len(sequence)+1-k):
3     kTails.state.append(x)

```

Now, the list variable *kTails.state* holds a set of states corresponding to the symbols. i.e [0,1,2,3,4,5,6,7]. For this particular example, we have seen earlier that states 0 and 4 are equivalent and so does states 1 and 5. So, the merging is done as follows: For each state in the list, we keep track of their equivalent relations and then replace the respective equivalent states with the state that has the lowest value. Other implementations might use strings to represent states such as s_0, s_1, \dots etc. Hence, in our example, we replace state 4 with 0 and 5 with 1. The list then becomes [0,1,2,3,0,1,6,7]. Since $k = 2$, we cut off the tail which includes states 6 and 7. Now, the final merged list becomes [0,1,2,3,0,1]. From this list, we define the state transitions as follows: $\{0 \xrightarrow{a} 1, 1 \xrightarrow{b} 2, 2 \xrightarrow{c} 3, 3 \xrightarrow{d} 0\}$. The automata representation of this set of transitions is given in Figure 2.2. The Python implementation of equivalent state checking and merging is given in Appendix 8.3. For the purpose of brevity, some statements in the code fragment have not been included.

4.2 Implementation of FSM Testing

We implemented test functionality in the tool which parses input protocol symbols based on a reference specification from which it learns the automata of the reference specification. When a set of input protocol symbols are processed against the reference FSM, it either rejects or accepts the input protocol symbols.

- Accept—A DFA is said to accept a particular input if and only if, starting in the start state and repeatedly applying the transition function to each input token in sequence, one winds up in a accepting state when one reaches the end of the input.
- Reject— Any input that is not accepted is rejected.

The language of a DFA is the set of all inputs it accepts; equivalently, the DFA is said to recognize that language. Our tool automatically rejects any non-deterministic automata. We implemented a DFA python module (dfa.py) in the kTail-PSM tool to capture the model of a set of reference protocol symbols. The module requires five parameters to be passed. These are: (1) *an initial state* (2) *accepting states* (3) *alphabet—a list of symbols* (4) *a set of states* (5) *a set of transitions*

To explore further, let take a look at the example discussed earlier in Section 4.1.2. By convention, we set initial state as 0.

```

1 start_state=0
2 accept_state={0}
3 alphabet={'a','b','c','d'}
4 states={0,1,2,3}
5 tf=dict()
6 tf[(0,'a')]=1
7 tf[(1,'b')]=2
8 tf[(2,'c')]=3
9 tf[(3,'d')]=0
10
11 d=DFA(states,alphabet,tf,start_state,accept_state)
12 inputstring=['a','b','c','x','d']
13 print d.run_with_input_list(inputstring)

```

Listing 4.3: FSM Testing implementation in Python

The variable *inputstring* holds the list of symbols to parse. If they are accepted against the reference specifications, the function returns *True* or *False* otherwise. In this case *inputstring* will not be accepted since there is an undefined symbol 'x'. However, the reference state machine will accept the following strings: ['a','b','c','d'] and an empty [] string since the accepting state is 0 which is also the initial state so it does not require an input symbol.

4.3 User Interface

To make it easier for users to interact with the kTail-PSM tool, we implemented graphical user interface (GUI). The GUI is implemented using Tkinter², which is Python's de-facto standard GUI package. A number of features are included in the GUI. This includes:

- loading protocol symbols from a text file
- a text area for the user to manually type or edit the protocol symbols
- a process log display screen that shows the necessary steps involved
- a canvas area that displays the protocol state machine

The GUI allows the user to interact with the tool and generate protocol state machines easily from the set of input symbols. Appendix 8.4 shows the GUI for the kTail-PSM tool.

4.4 Visualizing Automata

The protocol state automata is automatically generated once the input symbols are processed. To generate high-quality diagrams of the automata, we utilized the DOT³ program, a plain text graph description language that comes with the Graphviz⁴ package. When generating the FSM from the kTail-PSM tool, the data is rendered onto a image file called *../graph/ktail.png* in the package. The image is then loaded onto the canvas on the GUI using another function called *loadFSMImage()*. The code fragment below does the magic. It is implemented in the *ktail* module (*ktail.py*). Once graph is drawn to file, it is then loaded on the display canvas on the GUI.

²<https://wiki.python.org/moin/TkInter>

³<http://www.graphviz.org/content/dot-language>

⁴<http://www.graphviz.org/>

```

1 ktailFSM = FiniteStateMachine('K-TAIL')
2 ...
3 try:
4     graph=get_graph(ktailFSM)
5     if graph!=None:#Check if there is existing graph data
6         graph.draw('../graph/ktail.png', prog='dot')
7     else:
8         pass
9 except GraphvizError:
10     tkMessageBox.ERROR
11
12 def loadFSMImage():
13     try:
14         img = PhotoImage(file='../graph/ktail.png')
15         label.image = img # keep a reference!
16         imgWidth1 = canvas.winfo_width()
17         imgHeight1 = canvas.winfo_height()
18         x = (imgWidth1)/2.0
19         y = (imgHeight1)/2.0
20         return canvas.create_image(x, y, anchor=tk.CENTER,image=img,tags="bg_img")

```

Note that some statements in the code fragment are omitted for brevity. The kTail-PSM⁵ and Netzob v0.4.1 source⁶ codes are accessible from Github.com.

⁵kTail-PSM source code: <https://github.com/lnerit/ktailFSM>

⁶<https://github.com/lnerit/Netzob-0.4.1>

Chapter 5

Evaluation

This chapter presents the functional evaluation of the kTail-PSM tool. To ensure the correct operation of the implementation, the evaluation of the tool involved a number of different aspects.

5.1 Test Cases

One aspect of the evaluation of this project was focused on test cases to determine the correctness of functions in the source code. This involved writing unit tests and evaluating the code coverage.

5.1.1 Unit Testing

Unit testing refers to the practice of testing certain functions and areas – or units – of the code. This gives us the ability to verify that our functions work as expected. In other words, given a set of inputs, we can determine if the function is returning the proper values and will gracefully handle failures during the course of execution should invalid input be provided. To evaluate the functions implemented in kTail-PSM tool, we used Python unit testing framework referred to as *PyUnit*¹. The *unittest* module provides a rich set of tools for constructing and running tests.

A testcase is created by subclassing *unittest.TestCase*. Each test is defined with methods whose names start with the letters *test*. This naming convention informs the test runner about which methods represent tests.

In this project, we implemented 16 test cases. Each of the test cases were evaluated based on the following metrics: *input*, *expected output* and *result*. The result metric gives the status of the test case as a successful or unsuccessful when the test cases is executed. When the result of a test case is unsuccessful, the test case is reported as *Error* or a *Failure*.

The test cases are run in Eclipse using PyUnit. The test will execute some code and then use assert statements to check if the code executed correctly.

5.1.2 Code Coverage

To determine how much code was tested, we used Code Coverage² Python module. There are different types of code coverage metrics such as Statement coverage and Block coverage, Function coverage, Function call coverage and Branch coverage. It can be calculated using the formula:

¹<https://docs.python.org/2/library/unittest.html>

²<https://coverage.readthedocs.org/en/coverage-4.0.3/>

```

Let lines_of_tested_code = Number of lines of code exercised
Let total_lines=Total Number of lines of code
Then the code coverage is given by:
Code Coverage =(lines_of_tested_code)/(total_lines) * 100%

```

The code coverage for the kTail-PSM implementation is given Table 5.1.2

Module Name	Statements	Missed	Cover
dfa.py	53	36	32.1%
fsm.py	145	105	27.6%
gui.py	610	396	35.1%
State.py	48	48	0%
ktail.py	284	169	40.5%
resizeimage.py	17	9	47.1%
TOTAL	1157	763	30.1%

Table 5.1: Code Coverage for kTail-PSM tool implementation

Based on the coverage results, we only covered a less then 50% of overall coverage. The reason was that we only tested the important functions. Most of the helper functions were not covered in the test. Due to time constraint around this project, we were not able to cover test for all modules.

5.2 Existing Tools Evaluation

Before embarking on the task of extracting protocol symbols, firstly we evaluated the stable version of Netzob 0.4.1. This helped us to find the limitations in the tool as discussed next.

5.2.1 Tools Used

In this project, we used Netzob 0.4.1 to infer and extract protocol symbols. Since, it is an open-source project, there were a number of bugs that were not fixed. As a result, when importing network traces into the program, it was throwing exceptions. Hence, we debugged the buggy modules to correctly process the traces. The bugs were mostly associated with conversion errors of *int* and *float* data types. At first, we thought the tool was designed to cater for all types of protocols. So we spent a number of hours trying out different types of network protocol traces such as TCP, SMTP and POP3 protocols. This was simply to understand the features and functionalities of the tool. We also tried out the tool based on its demo presentations³ and its tutorials⁴. After trying out different types of protocols, we figured out the tool was mostly suitable for processing application-layer protocol payloads. We tried using the tool to extract TCP flags and then to infer its protocol state machine. However, the tool did not capture the required information. This indicated that the tool was not capable of extracting all protocol symbols from some binary protocols such as TCP.

Since Netzob has some limitations, we also used Wireshark 1.10.6 as a compliment. It is also an open-source, packet analysing tool that comes with allot of features as discussed in Section 3.3.2. We used Wireshark to extract PSM-related fields in the protocol specifications. In particular, we used a Lua plugin script for Wireshark, to extract PSM-related fields. The

³<https://www.youtube.com/watch?v=VYWFgKriaI0>

⁴https://dev.netzob.org/projects/netzob/wiki/Tutorial_getstarted

plugin script can be modified to extract specific fields to ones own requirements. In our case, we specified the script to extract the TCP flags while preserving the order and then we exported them to a comma-delimited text file which would later be imported into our kTail-PSM tool.

5.2.2 Symbol Extraction Format

For the application-layer protocol network traces, we used Netzob to extract the protocol symbols. We then utilized our customized export feature that we integrated into Netzob (refer to Section 3.3.1) to export the symbols to a text file format while preserving their order. The accessible path is: *File*→*Export Project*→*Human readable*.

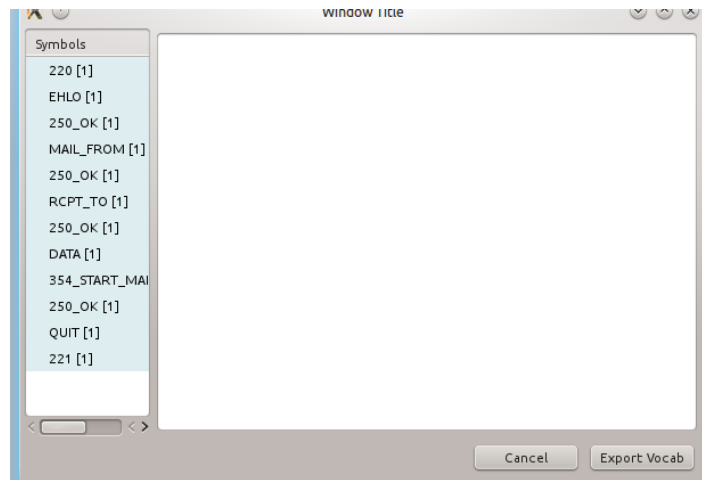


Figure 5.1: Shows customized export functionality integrated into Netzob

By clicking the *Export Vocab* button, we are able to save the symbols on the left to a text file in an ordered sequence. Figure 5.1 shows an ordered sequence of inferred SMTP protocol extracted using Netzob.

Similarly, for our specific case of extracting TCP flag codes in Wireshark, we used our customized *Lua* script to create a column in the trace display screen in Wireshark while maintaining the order of messages transmitted as shown in Figure 3.4. The symbols are then exported to a text file format. To achieve this, firstly, we exported the trace into a PCAP file. Then we used *tshark*⁵ (a packet capture tool that also has powerful reading and parsing features for PCAP analysis) to extract only the specific information. In this case, we extracted the TCP flag codes by executing the following script from a command line:

```
tshark -r SMTP.pcap -T fields -e tcpflags.flags > tcp_flags.txt
```

Listing 5.1: tshark script to extract TCP flags in an SMTP pcap file.

5.3 Trade-off between Precision and Generalisation based on k

After experimenting our tool with a number of different sequences of protocol symbols, it was observed that the k-Tail algorithm generalizes the behaviour from the observed sequences and it presents a promising technique for inferring behavioural models. To be specific, the k-Tail algorithm merges those states in the trace two next k invocations are

⁵<https://www.wireshark.org/docs/man-pages/tshark.html>

identical[43]. In other words, those states having equivalent k -Tails are merged as illustrated in Section 4.1.2. When selecting different values for k , it involves an intrinsic trade off between precision and generalization: — i.e when selecting smaller k implies more spurious merges with respect to precision and selecting larger k implies fewer merges with respect to generalization. Due to time limitations around this project, we only describe what we observed in executing the algorithm on a given sequence of protocol symbols with respect to precision and generalization of protocol state machines. More evaluation criteria should have been defined in this section to capture a minimized protocol state machine.

5.4 kTail-PSM average execution time

To evaluate the performance of the kTail-PSM tool, we selected different values of k and processed the sequence of symbols we extracted for the SMTP case study in Section 6.2.2. To minimize the validity of threats to the experiment, we made sure that there were no any other applications running except the system services on the testing computer. We recorded 30 execution readings for a given number of symbols in a sequence for a given k value. We then took the average value as the execution time and also calculated the standard deviation to determine the spread of the execution times. In the experiment, we selected k from 1 to 3. For each k , we started off with seven protocol symbols. We then incremented the sequence of symbols by a factor of seven and recorded the average time taken to process the automata (i.e ranging from 7 to 203 symbols). The average times that the tool takes to infer protocol state machine is presented in Figure 5.2.

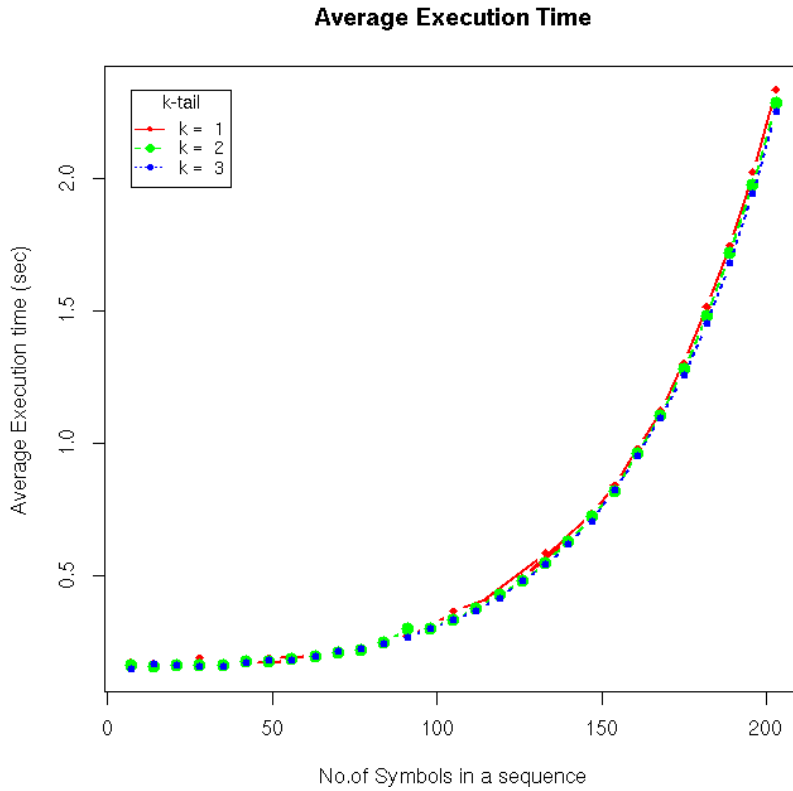


Figure 5.2: kTail-PSM average execution time

The kTail-PSM tool is able to process sequence having 200 protocol symbols in less than 2 seconds. We also note that time grows exponentially as the number of symbols increases,

k-value	No.of Symbols	mean	median
k=1	203	0.6312868225	0.0266295414
k=2	203	0.6171386998	0.0196360343
k=3	203	0.6078924618	0.0171054559
Combined	609	0.6188	0.0235

Table 5.2: Table showing the combined mean and distribution of execution times

which is due to merging steps of the methodology. The function of the execution time grows exponentially which can be expressed as $O(C^n)$ where C is a constant and n represents the number of symbols.

However, applying advance automata minimization algorithms could be utilized to further reduce the execution time for larger data sets. Furthermore, we also observe that when increasing k on a given sequence of symbols, average execution time drops. This is also due to merging steps of the methodology, since larger k means less number of merging steps and smaller k means more steps of merging. Figure 5.2 clearly shows this relationship. In any case, we were able to generate automata having less than 200 protocol symbols in a very short time.

Furthermore, to determine the degree of variation of time for different values of k , we measured the performance of our tool by determining the standard deviation of the execution times. That is to find out how far the individual execution times vary from the mean for different k values. We captured three execution time data sets as provided in Appendix 8.9. For each of the data sets we took, we combined them and calculated the overall average means and standard deviations as given in Table 8.9. From the result table, it can be seen that the standard deviation is very small with respect to the overall mean value. Hence, this means that in an ideal test environment, when different values of k is selected for the k-Tail algorithm execution on a sequence of symbols, there is a lower variation between the execution times. Again this is also captured in the average execution time graph in Figure 5.2 where the average execution times for three different k values are clustered together for a given number of protocol symbols.

Chapter 6

Case Study

In this chapter, we derive the protocol state machines from the kTail-PSM tool. Generally, the k-Tail algorithm works well for generating automata for simple sequences. This implies that it would not be suitable for inferring state machines for complex protocols. To put the tool to test, the protocol symbols of two different protocols were extracted and then fed into kTail-PSM tool. The sequence of symbols were then processed by applying the k-Tail algorithm and finally the protocol state machines were generated.

6.1 Experimenting kTail-PSM Tool

In this case study, we looked at two different protocols: – one binary protocol (TCP) and one text-based protocol (SMTP).

6.1.1 Data Sets

In the case study, we used normal publicly available SMTP and POP3 traces¹. The data sets contained messages exchanged between a single client and a single server. The sizes of both traces were less than 10 kilobytes. The network traces were obtained from a public repository in the Web, to facilitate the reproducibility of results, that is to preclude any bias to assist the reverse engineering task. To reverse engineer the the protocols from the traces, we assumed that traces were not encrypted. Hence, by extracting the protocol symbols with Netzob tool, the symbols were exported to the input format required by the kTail-PSM tool.

6.1.2 Testbed

The experiments with the kTail-PSM tool were carried out in a AMD E1-1500 APU 1480MHz with 4GB of memory running Ubuntu 14.04.3 LTS. The kTail-PSM tool is programmed in Python which takes in a comma-delimited sequence of protocol symbols. The tool uses the dot² program to generate high-quality diagrams of the automata.

6.2 Inferred Protocol State Machines (TCP/SMTP)

Since TCP and SMTP protocols are documented in RFC, the protocol state machines generated from our kTail-PSM tool are compared against the reference automata (manually produced from text description).

¹<http://asecuritysite.com/forensics>

²<http://www.graphviz.org/>

To infer the state machine for TCP protocol, we extracted the PSM-related field, which are the TCP flag codes: — SYN, ACK, PSH, RST and FIN flags. These codes are used in the session initialization process called 3-WAY *HANDSHAKE* between two hosts. The complete specification of the TCP protocol is defined in RFC 793³. To make it easier for the experiment, we focused on deriving client side of TCP protocol (an equivalent approach could be utilized for the server side), and therefore only the TCP flags sent from the client were used in the experiment. The TCP flags were extracted from the POP3 network trace file which contains 72 messages. However, since we filtered out the packets based on the client side, we were left with 24 messages. We observed that there were five different sessions established and each session was identified by the *SYN* flag in the sequence of flags extracted. The flags *SYN* and *ACKFIN* signify the start and end of a TCP session respectively. Since, the kTail-PSM tool only takes a single line of sequence, we combined all the flag codes extracted for each session in the order of their transmission as a single line of sequence with each symbol delimited by a comma. The combined TCP session flags list is given in Listing 6.1.

The PSMs were generated by selecting different values of k , and we chose the optimal value that generated an acceptable protocol state machine. We then compared our inferred PSM against the one defined in the RFC documentation.

SYN, ACK, ACKPSH, ACKPSH, ACK, ACKFIN, SYN, SYN, ACK, ACKPSH, ACKPSH, ACKPSH, ACKPSH, ACK, ACKFIN, SYN, ACK, ACKPSH, ACKPSH, ACKPSH, ACKPSH, ACK, ACKFIN, SYN, ACK, ACKPSH, ACKPSH, ACKPSH, ACKPSH, ACKPSH, ACKPSH, ACKPSH, ACKPSH, ACKPSH, ACKPSH, ACK, ACK, ACKPSH, ACK, ACK, ACK, ACKPSH, ACK, ACK, ACKPSH, ACK, ACKFIN

Listing 6.1: Sequence of TCP flags extracted.

The sequences of TCP flags in listing 6.1 was then imported into the kTail-PSM tool. By setting $k = 1$, we generated the inferred TCP protocol state machine as illustrated in Figure 6.1. Note that the TCP flags shown in the Figure 6.1 indicates what was sent for the transitions given upon receiving segments from the server. Note also in Listing 6.1, we combined response flags as a single word (such as *ACKPSH* and *ACKFIN*). The process log file is given in Appendix 8.7.

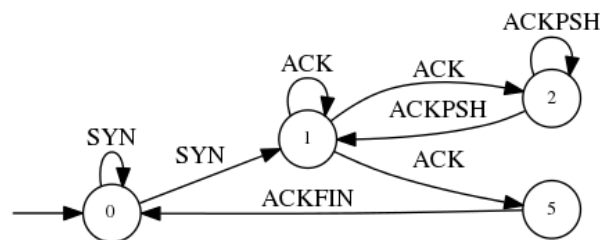


Figure 6.1: Inferred TCP client protocol state machine generated using kTail-PSM tool

In Figure 6.1, it can be seen that by applying the k-Tail algorithm, it identified four main states $\{0,1,2,5\}$ and the rest of the other states have been identified as equivalent states to these four states respectively and hence were merged. We compared this inferred state machine with the reference TCP client protocol state machine in Appendix 8.5. There are a few noticeable differences observed from the inferred one to that of the reference protocol state machine. For the inferred one, we derive the state machine based on the finite sequence of

³<https://tools.ietf.org/html/rfc793>

flags captured for each TCP session. Obviously, transitions triggered by symbols other than the flags were not captured. For instance, state transition based on *timeout* symbol was not taken into account as indicated in the reference protocol state machine.

However, it does capture the logic of the TCP protocol. At the initial State 0, the client initiated a connection by sending *SYN* symbol. There is also a transition to itself at State 0. This is acceptable since at States 6 and 7, the two transition symbols (*SYN*) sent are same and that these two states are equivalent to State 0 which have been merged with State 0. The reason being that there was a TCP retransmission due to time-out or transmission failure. When a *SYN* is acknowledged, the connection enters the *Connection established* state. State 1 in Figure 6.1, shows the connection established state. When the handshake is complete and until the closing begins at State 5, the connection is in *Data Transfer* state. This is captured at State 2 in the inferred protocol state machine. At State 2 the actual data was being exchanged. Based on the reference TCP client PSM, it does not indicate the data transmission state. Hence, it was captured in the inferred protocol state machine. At State 5, the client receives a *FIN* from the server and then sends *ACKFIN* transition and reverts to the initial state.

6.2.2 Inferred SMTP Protocol State Machine

Simple Mail Transfer Protocol (SMTP) is a protocol for sending e-mail messages between servers. Most e-mail systems that send mail over the Internet use SMTP to send messages from one server to another; the messages can then be retrieved with an e-mail client using either POP or IMAP server and the SMTP server when the e-mail application is configured.

For the inference of SMTP protocol state machine, we extracted the protocol symbols using Netzob. Then by using the customized export functionality that we integrated into Netzob, the symbols were exported into a comma-delimited text file format. Since SMTP is documented in the IETF RFC 5321⁴, this facilitates the comparison between the inferred automata and the reference automata (manually produced from the textual description). To make it easier for the experiment, we focussed on extracting protocol symbols from the *client* side. Here the *client* refers to the initiating server (the server originating the mail). In the SMTP data set, there were total of 62 messages. After filtering packets based on the *client* side, we obtained only 30 messages. However, we noted that these 30 messages also contained TCP session initialisation packets. So, we further filtered them out before importing them into Netzob. Hence, only the payload data was imported into Netzob. This was done using the Netzob trace import manager. This left us with 14 messages which were actually processed in Netzob. Listing 6.2 shows the extracted SMTP protocol symbols. Note that ordering has been preserved for the sequence of symbols given.

EHLO,MAIL.FROM,RCPT.TO,DATA,CONTENT,CRLF.CRLF,QUIT,EHLO,MAIL.FROM,RCPT.TO,DATA,CONTENT,CRLF.CRLF,QUIT

Listing 6.2: Sequence of SMTP protocol symbols extracted from Netzob.

While analysing the set of messages, we observed that hex code *0D0A2E0D0A*, appeared after the symbol *DATA*. This code translates to ASCII as: *0x0D* = *CR*, *0x0A* = *LF* and *0x2E* = *.* (dot) . The symbols were grouped into a single cluster when the clustering algorithm was applied in Netzob. Therefore, we represented them as one symbol, *CRLF.CRLF* as given in the sequence of symbols in Listing 6.2.

Similar to the approach taken in Section 6.2.1, we generated inferred protocol state machine by selecting different values of *k*, and then we chose an appropriate value of *k* that generated an acceptable automata compared to the reference automata. Again by setting

⁴<http://tools.ietf.org/html/rfc5321>

$k = 1$, we generate an acceptable inferred automata as given in Figure 6.2. The step-by-step processing log file is appended to Appendix 8.8.

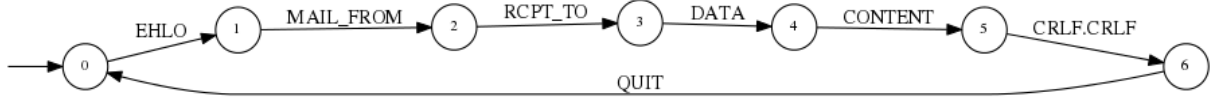


Figure 6.2: Inferred SMTP client protocol state machine generated using kTail-PSM tool

After executing k-Tail algorithm on the set of symbols, the final inferred SMTP PSM is shown in Figure 6.2. It is important to note that the PSM of SMTP protocol contains two parts, one is the state transition of client to server and the other is the state transition of server to client. In this experiment, we only focused on inferring state transition on the client to server state transition model.

From the inferred state machine, State 4 to State 5 only carry on the actual SMTP data. Hence, the SMTP data transmission does not contain any state information. This is followed by state transition information `<CRLF>.<CRLF>` as the end of mail data indication, which is defined in the RFC 5321. There are also instances that a mail is sent to multiple recipients. For such case, at State 2, it would have a state transition to itself. Such a scenario was not captured in the inferred state machine in Figure 6.2. The reason being that data set we used in this experiment only contained a single mail recipient. Furthermore, the self identification command captured in our inferred PSM is *EHLO*, which is extended *HELO* stated in the reference state machine. This is same as *HELO* but tells the server that the client may want to use the Extended SMTP (ESMTP) protocol instead. In comparison with the reference SMTP protocol state machine in Appendix 8.6, our inferred protocol state machine is equivalent to the reference automata.

Furthermore, we also compared our inferred SMTP state machine with a study done by Y.Wang, Z.Zhang, D.Yao et al.[15]. Our inferred state machine share similar state transitions except that theirs was based on probabilistic approach. Moreover, since the data set we used did not contain other SMTP commands such as *RSET*, it looks slightly different to the one inferred by Y.Wang, Z.Zhang, D.Yao et al. However, the logic captured in our inferred SMTP protocol state machine is equivalent to the probabilistic state machine that they produced.

Chapter 7

Conclusions

This chapter outlines the conclusions drawn and contributions made from the project as a whole, followed by future work. This project was set out with the aim of developing an automatic protocol state machine inference tool to complement the open-source protocol reverse engineering tool called Netzob to reverse engineer unknown network protocols.

7.1 Conclusions

A significant amount of research has been carried out in the area of communication protocol reverse engineering and various techniques to automatically extract protocol specifications. These includes grammatical inference, statistical analysis, keyword identification, static and dynamic analysis and other various approaches.

While most implementations of the approaches discussed have been focused on extracting protocol specifications, not much attention has been given to deriving protocol state machines. Of the protocol state machine implementation tools reviewed, none of them have been made available to industry domain. Most of the tools have been developed as a proof-of-concept tools in the academia. To help with protocol reverse engineering, an open-source tool called Netzob has been developed to reverse engineer communication protocols. The tool provides the functionality to extract protocol specifications from network traces. Furthermore, it provides the functionality to infer protocol grammar and simulate traffic flows.

However, after evaluating the tool, we discovered a number of limitations in the tool. So we implemented an open-source automatic PSM inference tool called ktail-PSM, that can be easily integrated into Netzob. We implemented the *k-tail* algorithm to achieve this. The tool takes an ordered sequence of protocol symbols extracted from Netzob and executes the *k-tail* algorithm on the sequence of symbols to identify equivalent relations among the given set of states. Those equivalent states identified are then merged. Finally, the merged states are transformed into an inferred protocol state machine.

We demonstrated our tools using protocol symbols extracted from real-world network traces for SMTP and TCP protocols. Based on the results produced from our case study analysis, the tool can infer PSMs to certain degree of precision depending on the value of k — smaller k means more spurious merges which tend to achieve higher precision and bigger k means less merges which tend to achieve higher generalization. There is a trade-off between them. In our case study, by choosing $k = 1$, the tool produced detailed PSMs for the two protocols when compared to the reference PSMs derived manually from their RFC specifications respectively. In terms of performance, ktail-PSM tool is able to process sequence having 200 protocol symbols in less than two seconds. We also found that time increases exponentially for longer sequences of input symbols. However, applying advanced automata

minimization algorithms could be utilized to further reduce the execution time for longer input sequences.

7.2 Contributions

In this report, we present the following contributions within the area of communication protocol reverse engineering, particularly in terms of reverse engineering communication protocols from network traces: (1) extended functionality in available open-source PRE tool (Netzob) to extract protocol specifications (symbols) from network traces and then export them into a comma-delimited text file format while preserving the ordering of the symbols (2) developed a tool (ktail-PSM) that automatically infers protocol state machines from an ordered set of given protocol symbols. Not only does the tool supports protocol symbols extracted from network-based traces but it can also be applied to sequence of symbols extracted from program-based traces. To achieve this, we implemented an equivalent state checking algorithm called the *k-tail* algorithm to identify equivalent state relations and merging them where necessary (3) we implemented FSM-based testing for an ordered input sequence of protocol symbols (4) showed its correctness by testing known protocols. (5) we conducted performance analysis on the implementation of the algorithm.

7.3 Future Work

The ktail-PSM tool takes in a sequence of protocol symbols extracted from Netzob as input and executes the k-Tail algorithm, automatically generating an inferred protocol state machine. However, the algorithm is not suited for handling complex protocols. Additionally, the tool is not fully optimized to handle multiple sequences of protocol symbols and it needs improvement. Taking these considerations into account, the following areas have been identified for future work:

- **Merging Multiple Sequence of Traces.** At the moment, the tool can handle a single sequence of protocol symbols. When there are multiple sequences, the tool simply concatenates them and forms a single sequence and infer the automata. Hence, for our future work, we will implement functionality to merge multiple canonical automata from different sequences of traces into a single minimized automata.
- **Optimization on Precision and Generalization of PSMs.** Another area that is left for future work is determining the precision of the inferred protocol state machine. This area obviously requires some techniques such as reduction and minimization to be applied to generate a PSM that can accept or reject any input protocol symbols.
- **Converting Non-Deterministic to Deterministic (NDFA) PSMs.** At this stage, the tool can infer both DFA and NDFA protocol state machines. An improvement to be added to it is to convert NDFA to a DFA protocol state machine.

Chapter 8

Appendix

8.1 Customized Export Functionality integrated into Netzob

```
1 def update(self):
2     global traceLog
3     self.view.symbolTreeview.get_model().clear()
4     self.traceLog=[]
5     for symbol in self.netzob.getCurrentProject().getVocabulary().getSymbols():
6         iter = self.view.symbolTreeview.get_model().append(None,
7             ["{0}".format(symbol.getID()), "{0} [{1} ".format(symbol.getName(),
8                 str(len(symbol.getMessages()))), '#000000', '#DEEEF0'])
9         self.traceLog.append(str(symbol))
10
11 def exportVocabToTextFile(self, fileName, vocabSymbol):
12     with open(str(fileName) + ".txt", "w") as trace_sequence:
13         trace_sequence.writelines("%s, " % item for item in vocabSymbol)
14
15 def locationToSaveFile(self, buttonclick):
16     try:
17         dialog = Gtk.FileChooserDialog("Save_Log", None,
18             Gtk.FileChooserAction.SAVE,
19             (Gtk.STOCK_CANCEL, Gtk.ResponseType.CANCEL,
20             Gtk.STOCK_SAVE, Gtk.ResponseType.ACCEPT))
21         response = dialog.run()
22         Gtk.FileChooser.set_do_overwrite_confirmation(dialog, True)
23         if response == Gtk.ResponseType.ACCEPT:
24             path = dialog.get_filename()
25             self.exportVocabToTextFile(path, self.traceLog)
26             dialog.destroy()
27             self.view.dialog.destroy()
28         elif response == Gtk.ResponseType.CANCEL:
29             dialog.destroy()
30     except IOError:
31         self.log=logging.getLogger(IOError)
```

8.2 Lua Script to extract TCP Flag codes

```
1 local function DecodeFlag(flags, mask, character)
2     if bit.band(flags, mask) == 0 then
3         return ''
4     else
5         return character
6     end
7 end
8
```

```

9 local function DefineAndRegisterTCPFlagsPostdissector()
10 local oProtoTCPFlags = Proto('tcpflags', 'TCP_Flags_Postdissector')
11 local oProtoFieldTCPFlags = ProtoField.string('tcpflags.flags',
12     'TCP_Flags', 'The_TCP_Flags')
13 oProtoTCPFlags.fields = {oProtoFieldTCPFlags}
14 local oField_tcp_flags = Field.new('tcp.flags')
15 function oProtoTCPFlags.dissector(buffer, pinfo, tree)
16     local i_tcp_flags = oField_tcp_flags()
17     local s_tcp_flags = ''
18     if i_tcp_flags ~= nil then
19         s_tcp_flags = s_tcp_flags .. DecodeFlag(i_tcp_flags.value, 0x80, 'C')
20         s_tcp_flags = s_tcp_flags .. DecodeFlag(i_tcp_flags.value, 0x40, 'E')
21         s_tcp_flags = s_tcp_flags .. DecodeFlag(i_tcp_flags.value, 0x20, 'URG')
22         s_tcp_flags = s_tcp_flags .. DecodeFlag(i_tcp_flags.value, 0x10, 'ACK')
23         s_tcp_flags = s_tcp_flags .. DecodeFlag(i_tcp_flags.value, 0x08, 'PSH')
24         s_tcp_flags = s_tcp_flags .. DecodeFlag(i_tcp_flags.value, 0x04, 'RST')
25         s_tcp_flags = s_tcp_flags .. DecodeFlag(i_tcp_flags.value, 0x02, 'SYN')
26         s_tcp_flags = s_tcp_flags .. DecodeFlag(i_tcp_flags.value, 0x01, 'FIN')
27         local oSubtree = tree:add(oProtoTCPFlags, 'TCP_Flags')
28         oSubtree:add(oProtoFieldTCPFlags, s_tcp_flags)
29     end
30 end
31 register_postdissector(oProtoTCPFlags)
32
33 end
34 local function Main()
35     DefineAndRegisterTCPFlagsPostdissector()
36 end
37
38 Main()

```

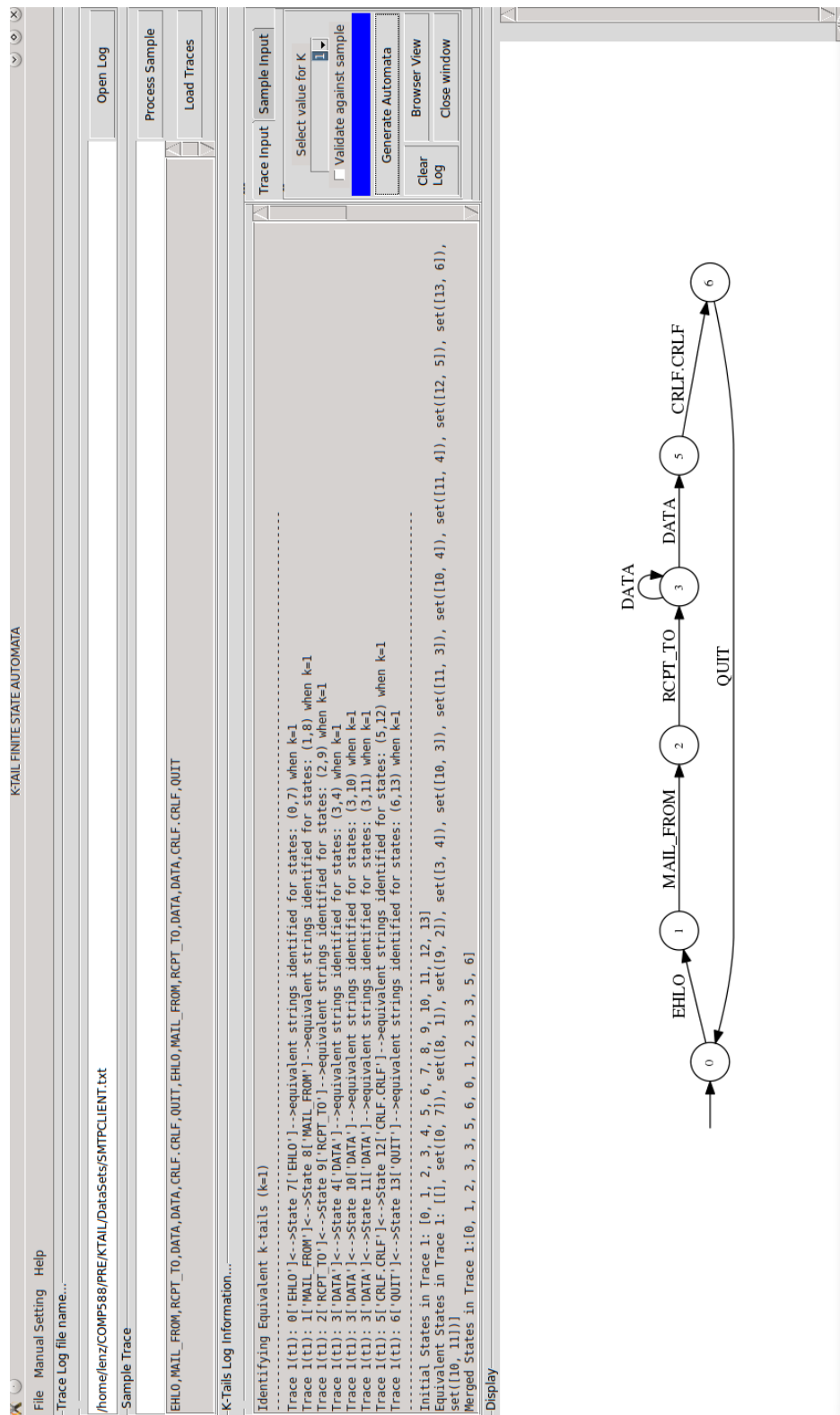
8.3 Implementation of k-Tail Equivalent State checking and Merging them

```

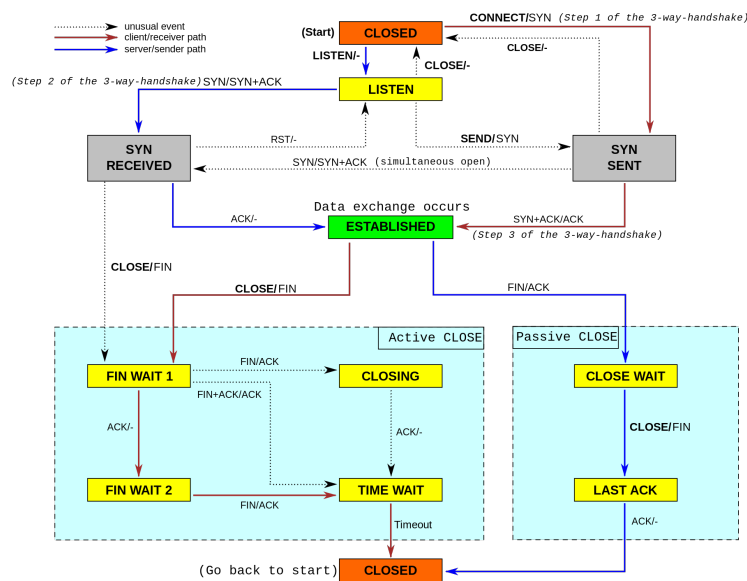
1 def do_kTailEquivCheck(self, k, seq):
2     assert(k>0)
3     sequence=[]
4     kTails.state=[]
5     kTails.mergedlist=[]
6     sequence=seq
7
8     for x in range(0, len(sequence)+1-k):
9         kTails.state.append(x)
10
11     for i in range(0, len(kTails.state)):
12         for ind in kTails.state:
13             #check that the next sequence of k-length strings is not empty
14             #Here assume that the order of the sequence is important
15             if (len(sequence[ind+1+i:ind+k+1+i])<k):
16                 pass
17             elif ind==None:
18                 pass
19             else:
20                 if check_equivalence(sequence[i:k+i], sequence[ind+1+i:ind+k+1+i]):
21                     if i in kTails.mergedlist:
22                         kTails.mergedlist[ind+i+1]=i

```

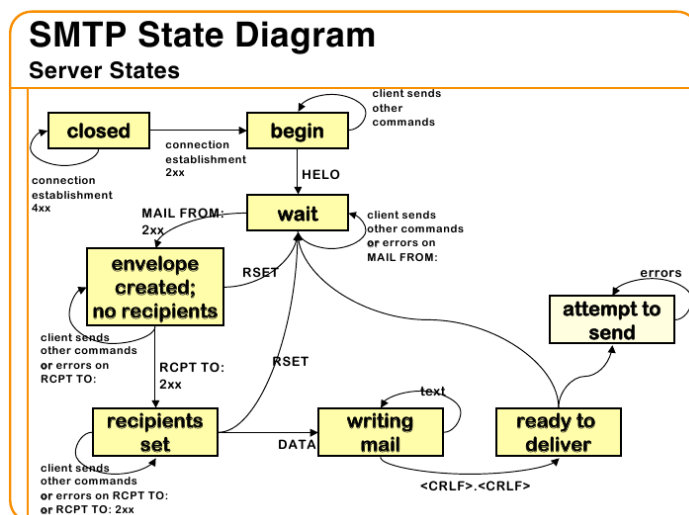
8.4 kTail-PSM tool Graphical User Interface



8.5 TCP Reference Protocol State Machine¹



8.6 Simplified SMTP Reference Protocol State Machine²



¹https://en.wikipedia.org/wiki/Transmission_Control_Protocol

²<http://denis.papathanasiou.org/posts/2011.11.11.post.html>

8.7 Inferred TCP Protocol State Machine Processed Log

Symbol Sequence: [SYN, ACK, ACKPSH, ACKPSH, ACK, ACKFIN, SYN, SYN, ACK, ACKPSH, ACKPSH, ACKPSH, ACKPSH, ACK, ACKFIN, SYN, ACK, ACKPSH, ACKPSH, ACKPSH, ACKPSH, ACK, ACKFIN, SYN, ACK, ACKPSH, ACKPSH, ACKPSH, ACKPSH, ACKPSH, ACKPSH, ACKPSH, ACK, ACK, ACKPSH, ACK, ACK, ACK, ACKPSH, ACK, ACK, ACKPSH, ACK, ACKFIN]

Identifying Equivalent k-tails (k=1)

```
0[ 'SYN' ]<-->State 6[ 'SYN' ]-->equivalent strings identified for states: (0,6)
0[ 'SYN' ]<-->State 7[ 'SYN' ]-->equivalent strings identified for states: (0,7)
0[ 'SYN' ]<-->State 15[ 'SYN' ]-->equivalent strings identified for states: (0,15)
0[ 'SYN' ]<-->State 21[ 'SYN' ]-->equivalent strings identified for states: (0,21)
0[ 'SYN' ]<-->State 30[ 'SYN' ]-->equivalent strings identified for states: (0,30)
1[ 'ACK' ]<-->State 4[ 'ACK' ]-->equivalent strings identified for states: (1,4)
1[ 'ACK' ]<-->State 8[ 'ACK' ]-->equivalent strings identified for states: (1,8)
1[ 'ACK' ]<-->State 13[ 'ACK' ]-->equivalent strings identified for states: (1,13)
1[ 'ACK' ]<-->State 16[ 'ACK' ]-->equivalent strings identified for states: (1,16)
1[ 'ACK' ]<-->State 19[ 'ACK' ]-->equivalent strings identified for states: (1,19)
1[ 'ACK' ]<-->State 22[ 'ACK' ]-->equivalent strings identified for states: (1,22)
1[ 'ACK' ]<-->State 28[ 'ACK' ]-->equivalent strings identified for states: (1,28)
1[ 'ACK' ]<-->State 31[ 'ACK' ]-->equivalent strings identified for states: (1,31)
1[ 'ACK' ]<-->State 40[ 'ACK' ]-->equivalent strings identified for states: (1,40)
1[ 'ACK' ]<-->State 41[ 'ACK' ]-->equivalent strings identified for states: (1,41)
1[ 'ACK' ]<-->State 43[ 'ACK' ]-->equivalent strings identified for states: (1,43)
1[ 'ACK' ]<-->State 44[ 'ACK' ]-->equivalent strings identified for states: (1,44)
1[ 'ACK' ]<-->State 45[ 'ACK' ]-->equivalent strings identified for states: (1,45)
1[ 'ACK' ]<-->State 47[ 'ACK' ]-->equivalent strings identified for states: (1,47)
1[ 'ACK' ]<-->State 48[ 'ACK' ]-->equivalent strings identified for states: (1,48)
1[ 'ACK' ]<-->State 50[ 'ACK' ]-->equivalent strings identified for states: (1,50)
2[ 'ACKPSH' ]<-->State 3[ 'ACKPSH' ]-->equivalent strings identified for states: (2,3)
2[ 'ACKPSH' ]<-->State 9[ 'ACKPSH' ]-->equivalent strings identified for states: (2,9)
2[ 'ACKPSH' ]<-->State 10[ 'ACKPSH' ]-->equivalent strings identified for states: (2,10)
2[ 'ACKPSH' ]<-->State 11[ 'ACKPSH' ]-->equivalent strings identified for states: (2,11)
2[ 'ACKPSH' ]<-->State 12[ 'ACKPSH' ]-->equivalent strings identified for states: (2,12)
2[ 'ACKPSH' ]<-->State 17[ 'ACKPSH' ]-->equivalent strings identified for states: (2,17)
2[ 'ACKPSH' ]<-->State 18[ 'ACKPSH' ]-->equivalent strings identified for states: (2,18)
2[ 'ACKPSH' ]<-->State 23[ 'ACKPSH' ]-->equivalent strings identified for states: (2,23)
2[ 'ACKPSH' ]<-->State 24[ 'ACKPSH' ]-->equivalent strings identified for states: (2,24)
2[ 'ACKPSH' ]<-->State 25[ 'ACKPSH' ]-->equivalent strings identified for states: (2,25)
2[ 'ACKPSH' ]<-->State 26[ 'ACKPSH' ]-->equivalent strings identified for states: (2,26)
2[ 'ACKPSH' ]<-->State 27[ 'ACKPSH' ]-->equivalent strings identified for states: (2,27)
2[ 'ACKPSH' ]<-->State 32[ 'ACKPSH' ]-->equivalent strings identified for states: (2,32)
2[ 'ACKPSH' ]<-->State 33[ 'ACKPSH' ]-->equivalent strings identified for states: (2,33)
2[ 'ACKPSH' ]<-->State 34[ 'ACKPSH' ]-->equivalent strings identified for states: (2,34)
2[ 'ACKPSH' ]<-->State 35[ 'ACKPSH' ]-->equivalent strings identified for states: (2,35)
2[ 'ACKPSH' ]<-->State 36[ 'ACKPSH' ]-->equivalent strings identified for states: (2,36)
2[ 'ACKPSH' ]<-->State 37[ 'ACKPSH' ]-->equivalent strings identified for states: (2,37)
2[ 'ACKPSH' ]<-->State 38[ 'ACKPSH' ]-->equivalent strings identified for states: (2,38)
2[ 'ACKPSH' ]<-->State 39[ 'ACKPSH' ]-->equivalent strings identified for states: (2,39)
2[ 'ACKPSH' ]<-->State 42[ 'ACKPSH' ]-->equivalent strings identified for states: (2,42)
2[ 'ACKPSH' ]<-->State 46[ 'ACKPSH' ]-->equivalent strings identified for states: (2,46)
2[ 'ACKPSH' ]<-->State 49[ 'ACKPSH' ]-->equivalent strings identified for states: (2,49)
5[ 'ACKFIN' ]<-->State 14[ 'ACKFIN' ]-->equivalent strings identified for states: (5,14)
5[ 'ACKFIN' ]<-->State 20[ 'ACKFIN' ]-->equivalent strings identified for states: (5,20)
5[ 'ACKFIN' ]<-->State 29[ 'ACKFIN' ]-->equivalent strings identified for states: (5,29)
5[ 'ACKFIN' ]<-->State 51[ 'ACKFIN' ]-->equivalent strings identified for states: (5,51)
```

Initial States in Trace: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36,

37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51]

Merged States in Trace:[0, 1, 2, 2, 1, 5, 0, 0, 1, 2, 2, 2, 2, 1, 5, 0, 1, 2, 2, 1, 5, 0, 1, 2, 2, 2, 2, 1, 5, 0, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1, 1, 2, 1, 1, 1, 2, 1, 1, 2, 1, 5]

State Map Dictionary in Trace 1: {0: {0: 'SYN', 7: 'SYN'}, 8: {8: 'ACKPSH', 7: 'ACKPSH'}, 11: {0: 'ACKFIN'}, 7: {8: 'ACK', 11: 'ACK', 7: 'ACK'}}

Finalised States in Trace 1: **set**([0, 1, 2, 5])

+++++

State-Label in Trace 1: {0: 'SYN', 1: 'ACK', 2: 'ACKPSH', 5: 'ACKFIN'}

State Transitions:

0-->0[label=SYN]

0-->1[label=SYN]

1-->1[label=ACK]

1-->2[label=ACK]

1-->5[label=ACK]

2-->1[label=ACKPSH]

2-->2[label=ACKPSH]

5-->0[label=ACKFIN]

+++++

8.8 Inferred SMTP Protocol State Machine Processed Log

Symbol Sequence: [EHLO,MAILFROM,RCPT.TO,DATA,CONTENT,CRLF.CRLF,QUIT,EHLO,MAILFROM,RCPT.TO,DATA,CONTENT,CRLF.CRLF,QUIT]

Identifying Equivalent k-tails (k=1)

0['EHLO']<-->State 7['EHLO']-->equivalent strings identified for states: (0,7)
1['MAILFROM']<-->State 8['MAILFROM']-->equivalent strings identified for states: (1,8)
2['RCPT.TO']<-->State 9['RCPT.TO']-->equivalent strings identified for states: (2,9)
3['DATA']<-->State 10['DATA']-->equivalent strings identified for states: (3,10)
4['CONTENT']<-->State 11['CONTENT']-->equivalent strings identified for states: (4,11)
5['CRLF.CRLF']<-->State 12['CRLF.CRLF']-->equivalent strings identified for states: (5,12)
6['QUIT']<-->State 13['QUIT']-->equivalent strings identified for states: (6,13)

Initial States in Trace: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13]
Equivalent States in Trace: [[], set([0, 7]), set([8, 1]), set([9, 2]), set([10, 3]), set([11, 4]), set([12, 5]), set([13, 6])]
Merged States in Trace:[0, 1, 2, 3, 4, 5, 6, 0, 1, 2, 3, 4, 5, 6]
Mapping in Trace 1: ['0-->1', '1-->2', '2-->3', '3-->4', '4-->5', '5-->6', '6-->0', '0-->1', '1-->2', '2-->3', '3-->4', '4-->5', '5-->6']
State Map Dictionary in Trace: {0: {1: 'EHLO'}, 1: {2: 'MAILFROM'}, 2: {3: 'RCPT.TO'}, 3: {4: 'DATA'}, 4: {5: 'CONTENT'}, 5: {6: 'CRLF.CRLF'}, 6: {0: 'QUIT'}}
Finalised States in Trace: set([0, 1, 2, 3, 4, 5, 6])
+++++
State-Label in Trace: {0: 'EHLO', 1: 'MAILFROM', 2: 'RCPT.TO', 3: 'DATA', 4: 'CONTENT', 5: 'CRLF.CRLF', 6: 'QUIT'}

State Transitions:

0-->1[label=EHLO]
1-->2[label=MAILFROM]
2-->3[label=RCPT.TO]
3-->4[label=DATA]
4-->5[label=CONTENT]
5-->6[label=CRLF.CRLF]
6-->0[label=QUIT]

+++++

8.9 Execution time data sets for ktail-PSM tool

k-value	No.Of Symbols	mean	median
1	7	0.170305275917	0.0766093897365
1	14	0.165733591715	0.0250547104895
1	21	0.165936279297	0.0193765955892
1	28	0.185439817111	0.0342942193741
1	35	0.168108622233	0.0202403146662
1	42	0.168099514643	0.013277894782
1	56	0.17844089667	0.00880253951171
1	49	0.185798374812	0.0314015313365
1	63	0.195016781489	0.0229220970447
1	70	0.204019077619	0.0149879020834
1	77	0.223977255821	0.0148997617271
1	84	0.24873667558	0.0272690664674
1	91	0.270370618502	0.00980755272035
1	98	0.304340195656	0.0225289744164
1	105	0.362445958455	0.118474921274
1	119	0.42706849575	0.0142487980826
1	112	0.382620127996	0.0175744876151
1	133	0.585056130091	0.0809484640039
1	140	0.635286903381	0.0134015498699
1	126	0.488477055232	0.0107600898287
1	147	0.733863647779	0.0132811507015
1	154	0.838589088122	0.00987970492867
1	161	0.97818924586	0.0176338522227
1	168	1.12445046107	0.017658052563
1	175	1.30035101573	0.0269539111483
1	182	1.51391485532	0.0221093479558
1	189	1.74465762774	0.0210587057845
1	196	2.02399130662	0.0230960074552
1	203	2.33403295676	0.0237051077397

Table 8.1: Data set 1 for $k = 1$

k-value	No.Of Symbols	mean	median
2	7	0.158611369133	0.0336188402759
2	14	0.155111996333	0.0108064201206
2	21	0.159634868304	0.0135385392822
2	28	0.159431918462	0.0121031978083
2	35	0.160711097717	0.00936100152164
2	42	0.17304623127	0.019418705553
2	49	0.172278952599	0.0150207859594
2	56	0.182353027662	0.0147811016568
2	63	0.191815201441	0.017991946285
2	70	0.20451742808	0.012632717902
2	77	0.216569892565	0.0118841202075
2	84	0.243025970459	0.0123340900345
2	91	0.297150158882	0.104006554199
2	98	0.299482425054	0.0271416596276
2	105	0.330850601196	0.0144353390817
2	112	0.372930622101	0.0158886030086
2	119	0.424441512426	0.0133603587245
2	126	0.480536897977	0.0142375061442
2	133	0.546192709605	0.0114398504989
2	140	0.625015266736	0.0146539751593
2	147	0.719621984164	0.0146008423735
2	154	0.818739899	0.0117643422667
2	161	0.960800361633	0.0141455758316
2	168	1.10440553029	0.0205973251558
2	175	1.28172816435	0.030259980873
2	182	1.48002635638	0.0241718422309
2	189	1.71826092402	0.0164496100933
2	196	1.97648781935	0.0173127124357
2	203	2.2832431078	0.021487451748

Table 8.2: Data set 2 for $k = 2$

k-value	No.Of Symbols	mean	median
3	7	0.145054626465	0.0154363363395
3	14	0.16574280262	0.0223905246324
3	21	0.157617791494	0.00884266219517
3	28	0.15617860953	0.00998829017736
3	35	0.156658864021	0.00606182231144
3	42	0.170666201909	0.0243893654738
3	49	0.176795450846	0.0175591073877
3	56	0.17878613472	0.0115497967542
3	63	0.190685629845	0.0109987660047
3	70	0.211542574565	0.0257091739797
3	77	0.219898509979	0.0109690054913
3	84	0.239225522677	0.0183752721525
3	91	0.263620495796	0.0132567331783
3	98	0.297256032626	0.019203016819
3	105	0.329963199298	0.0154315186468
3	112	0.365391755104	0.0110427999707
3	119	0.414150746663	0.0122175198031
3	126	0.477417151133	0.0177525007413
3	133	0.541635028521	0.0107335964917
3	140	0.615150149663	0.012331360349
3	147	0.704394586881	0.0117758920167
3	154	0.823501650492	0.0193841043156
3	161	0.952754100164	0.0166473840231
3	168	1.09466435909	0.0415337716205
3	175	1.25593523184	0.0228552500434
3	182	1.45092353026	0.0211433200663
3	189	1.67766602834	0.0212738055831
3	196	1.94322921435	0.0185145362915
3	203	2.25237541199	0.028690988024

Table 8.3: Data set 3 for $k = 3$

Bibliography

- [1] G. Bossert and F. Guihéry, “Security Evaluation of Communication Protocols in Common Criteria,” *International Common Criteria Conference*, 2012.
- [2] X. Ming-Ming and Y. Shun-Zheng, “Recovering Models of Network Protocol Using Grammatical Inference,” *Procedia Engineering*, vol. 15, pp. 3764–3768, 2011.
- [3] Y.-H. Shin and E.-G. Im, “A Survey of Recently Automatic Protocol Reverse Engineering Mechanisms,” , pp. 1185–1187.
- [4] M.-M. Xiao, S.-Z. Yu, and Y. Wang, “Automatic Network Protocol Automaton Extraction,” in *2009 Third International Conference on Network and System Security*, pp. 336–343, IEEE, 2009.
- [5] Y. Wang, N. Zhang, Y. Wu, and B. Su, “Protocol Specification Inference Based on Keywords Identification,” *Advanced Data Mining and Applications*, 2013.
- [6] A. Biermann and J. Feldman, “On the synthesis of finite-state machines from samples of their behavior,” *Computers, IEEE Transactions on*, 1972.
- [7] G. Bossert, “Exploiting Semantic for the Automatic Reverse Engineering of Communication Protocols,” 2014.
- [8] Z. Zhang, Q.-Y. Wen, and W. Tang, “Mining Protocol State Machines by Interactive Grammar Inference,” in *2012 Third International Conference on Digital Manufacturing & Automation*, pp. 524–527, IEEE, jul 2012.
- [9] “Skype Reverse Engineering : The (long) journey ;).. — oK Labs.”
- [10] J. Narayan, S. Shukla, and T. Clancy, “A Survey of Automatic Protocol Reverse Engineering Tools,” *ACM Computing Surveys (CSUR)*, 2015.
- [11] R. Graham and P. Johnson, “Finite state machine parsing for internet protocols: Faster than you think,” *Security and Privacy Workshops (...)*, 2014.
- [12] A. D’Ulizia, F. Ferri, and P. Grifoni, “A survey of grammatical inference methods for natural language learning,” *Artificial Intelligence Review*, 2011.
- [13] M. Xiao and S. Yu, “Learning automata representation of network protocol by grammar induction,” *Web Information Systems and Mining*, 2010.
- [14] Fanzhi Meng, Yuan Liu, Chunrui Zhang, Tong Li, and Yang Yue, “Inferring protocol state machine for binary communication protocol,” in *2014 IEEE Workshop on Advanced Research and Technology in Industry Applications (WARTIA)*, pp. 870–874, IEEE, sep 2014.
- [15] Y. Wang, Z. Zhang, D. Yao, B. Qu, and L. Guo, “Inferring protocol state machine from network traces: a probabilistic approach,” ... *Cryptography and Network ...*, 2011.

- [16] Y. Wang, X. Li, J. Meng, Y. Zhao, Z. Zhang, and L. Guo, "Biprominer: Automatic Mining of Binary Protocol Features," in *2011 12th International Conference on Parallel and Distributed Computing, Applications and Technologies*, pp. 179–184, IEEE, oct 2011.
- [17] J. Caballeroa and D. Songb, "Automatic Protocol Reverse-Engineering: Message Format Extraction and Field Semantics Inference," *cs.berkeley.edu*.
- [18] W. Cui, J. Kannan, and H. Wang, "Discoverer: Automatic Protocol Reverse Engineering from Network Traces.," *USENIX Security*, 2007.
- [19] J. Luo and S. Yu, "Position-based automatic reverse engineering of network protocols," *Journal of Network and Computer Applications*, 2013.
- [20] N. Nethercote, "Dynamic binary analysis and instrumentation," 2004.
- [21] M. Shevertalov and S. Mancoridis, "A Reverse Engineering Tool for Extracting Protocols of Networked Applications," in *14th Working Conference on Reverse Engineering (WCRE 2007)*, pp. 229–238, IEEE, oct 2007.
- [22] Z. Lin, X. Jiang, D. Xu, and X. Zhang, "Automatic Protocol Format Reverse Engineering through Context-Aware Monitored Execution.," *NDSS*, 2008.
- [23] W. Cui, M. Peinado, and K. Chen, "Tupni: Automatic reverse engineering of input formats," *Proceedings of the 15th ...*, 2008.
- [24] J. Caballero and P. Poosankam, "Dispatcher: Enabling active botnet infiltration using automatic protocol reverse-engineering," *Proceedings of the 16th ...*, 2009.
- [25] P. Comparetti, G. Wondracek, C. Kruegel, and E. Kirda, "Prospex: Protocol Specification Extraction," *mail.seclab.tuwien.ac.at*.
- [26] Z. Lin, X. Zhang, and D. Xu, "Automatic reverse engineering of data structures from binary execution," 2010.
- [27] J. Caballero, H. Yin, Z. Liang, and D. Song, "Polyglot: Automatic extraction of protocol message format using dynamic binary analysis," ... of the 14th ACM conference on ..., 2007.
- [28] Y. Wang, N. Zhang, Y.-m. Wu, B.-b. Su, and Y.-j. Liao, "Protocol Formats Reverse Engineering Based on Association Rules in Wireless Environment," in *2013 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications*, pp. 134–141, IEEE, jul 2013.
- [29] Z. Wang, X. Jiang, W. Cui, X. Wang, and M. Grace, "ReFormat: Automatic reverse engineering of encrypted messages," *Computer SecurityESORICS ...*, 2009.
- [30] J. Newsome and D. Song, "Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software," vol. 44, pp. 2–3, 2005.
- [31] Y. He, H. Shu, and X. Xiong, "Protocol Reverse Engineering Based on DynamoRIO," in *2009 International Conference on Information and Multimedia Technology*, pp. 310–314, IEEE, 2009.
- [32] Y. Wang, X. Yun, and M. Shafiq, "A semantics aware approach to automated reverse engineering unknown protocols," *Network Protocols (...)*, 2012.

- [33] S. Whalen, M. Bishop, and J. Crutchfield, "Hidden markov models for automated protocol learning," *Security and Privacy in ...*, 2010.
- [34] Z. Lin, X. Jiang, D. Xu, and X. Zhang, "Automatic Protocol Format Reverse Engineering through Context-Aware Monitored Execution," in *15Th Symposium on Network and Distributed System Security*, p. 46, 2008.
- [35] Y. Wang, M. Z. Shafiq, L. Wang, A. X. Liu, Z. Zhang, D. Yao, Y. Zhang, and L. Guo, "A semantics aware approach to automated reverse engineering unknown protocols," in *2012 20th IEEE International Conference on Network Protocols (ICNP)*, pp. 1–10, IEEE, oct 2012.
- [36] S. Needleman and C. Wunsch, "A general method applicable to the search for similarities in the amino acid sequence of two proteins," *Journal of molecular biology*, 1970.
- [37] W. Ying, L. GU, Z. LI, and Y. YANG, "Protocol reverse engineering through dynamic and static binary analysis," *The Journal of China Universities of Posts ...*, 2013.
- [38] C. Leita, K. Mermoud, and M. Dacier, "Scriptgen: an automated script generation tool for honeyd," ... *Applications Conference, 21st ...*, 2005.
- [39] N. Provos, "A Virtual Honeypot Framework.," *USENIX Security Symposium*, 2004.
- [40] J. Antunes, N. Neves, and P. Verissimo, "ReverX: Reverse Engineering of Protocols," 2011.
- [41] I. Beschastnikh and Y. Brun, "Unifying FSM-inference algorithms through declarative specification," *Proceedings of the ...*, 2013.
- [42] F. Schneider, "Enforceable security policies," *ACM Transactions on Information and System Security ...*, 2000.
- [43] I. Krka, Y. Brun, and D. Popescu, "Using dynamic execution traces and program invariants to enhance behavioral model inference," ... , *2010 ACM/IEEE ...*, 2010.