Konstantina Kanellopoulou

Active Automata Learning for Network Protocols

Master's Thesis

Graz University of Technology

Institute of Applied Information Processing and Communications
Head: Univ.-Prof. Dipl.-Ing. Dr.techn Reinhard Posch

Supervisor: Univ.-Prof. Ph.D. Roderick Bloem

Graz, November 2016

Statutory Declaration

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

Graz, ______________________________ ______________________________

Date Signature

Eidesstattliche Erklärung[1]

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Graz, am ______________________________ ______________________________

Datum Unterschrift

[1]Beschluss der Curricula-Kommission für Bachelor-, Master- und Diplomstudien vom 10.11.2008; Genehmigung des Senates am 1.12.2008

Acknowledgements

I am indebted to my colleagues at the IIAK who have provided invaluable help and feedback during the course of my work.

I especially wish to thank my advisor, Franz Roeck, for his immediate attention to my questions and endless hours of toil in correcting draft versions of this thesis.

Special mention goes to Denise for her help in supporting me and for ample supplies of coffee.

Last but not least, without the support and understanding of my partner Christof and the tolerance of my friends, this thesis would not have been possible.

Konstantina Kanellopoulou
Graz, Austria, April 2016

Abstract

In this thesis we have analyzed two network protocols by using automata learning techniques in order to observe their behaviour and discover possible unexpected system reactions. We have focused on the behaviour of the protocols in relation to their initial specifications. The first protocol which we have analyzed is an extension of the SSL protocol, known as Heartbeat Protocol [1]. The second protocol which we evaluated is the File Transfer Protocol (FTP) [19] and specifically three client implementations of it. Moreover, we have conducted a comparison between the different implementations of the file transfer protocol specification. For our experiments we have used a regular inference tool, called LearnLib, for applying the automata learning techniques. In order to address performance and functionality issues, we have used a mapper during the learning phase. The mapper performs a mapping from concrete sets of inputs/outputs to abstract ones and vice versa. In that way, the learning tool sends abstract input/output sets to the mapper, which concretises them and sends them to the the system under learn. We show that we could not learn the behavior of the Heartbeat protocol and to represent a model of it when using this learning setup presented in this thesis. On the other hand, we were able to learn successfully and construct a model for the three client implementations of the FTP protocol. However, we demonstrate that the different implementations of the FTP protocol generate different automata-learned models and some of them do not comply to the RFC model specification. Moreover, the FTP client from Apache shows unexpected behavior when specific input traces are triggered. Thus, active automata learning techniques can be used in real life scenarios and applications for testing and evaluating the behaviour of the target systems.

Contents

Acknowledgements v

Abstract vii

1 Introduction 1

1.1 Motivation . . . 2

1.1.1 Why correctness of a software system is important and what is the current situation . . . 2

1.1.2 Network Protocols - Incorrect implementations lead to security vulnerabilities . . . 3

1.2 Our Approach to the problem . . . 5

1.3 Background knowledge of the Learning Theory . . . 12

1.3.1 Machine Learning & the three basic types of Learning 13

1.3.2 From Machine Learning to Automata Learning . . . 14

1.4 Overview and organisation of this thesis . . . 15

2 Preliminaries and Basic Concepts of Automata Learning 17

2.1 Passive and Active Automata learning . . . 21

2.1.1 Angluin's L* Algorithm . . . 23

2.2 Black Box System . . . 27

3 Active Learning Tools 29

3.1 LearnLib . . . 29

3.1.1 The MAT model interfaces in LearnLib . . . 35

3.1.2 Mapper . . . 37

3.1.3 Test Adapter . . . 39

3.2 LearnLib Studio . . . 39

3.3 Tomte . . . 41

4 Network Protocols **43**
4.1 File Transfer Protocol 43
4.2 Heartbeat Protocol 50

5 Active Learning of Network Protocols **57**
5.1 Heartbeat Protocol 60
5.1.1 Mapper for Heartbeat Protocol 61
5.2 FTP Protocol 66
5.2.1 Server Setup 68
5.2.2 Client Setup 69
5.3 Issues and Tunings 74

6 Experimental Results and Discussion **77**
6.1 Heartbeat protocol 78
6.1.1 Results 78
6.1.2 Discussion 79
6.2 FTP Protocol 81
6.2.1 Results, common in all FTP Implementations 81
6.2.2 SimpleFTP 85
6.2.3 FTP4j Client 87
6.2.4 FTPClient Apache 90
6.3 Benchmarking 94

7 Related Work **97**

8 Conclusions and Future Work **103**

Bibliography **111**

List of Figures

1.1 Test Adapter . . . 8

2.1 Example of an automaton with state *e* as accepting state . . . 18
2.2 The prefix tree acceptor for {baa, baba} . . . 22
2.3 Components of the L* Algorithm . . . 25

3.1 The MAT Model components as related to the LearnLib interfaces (dashed borders), source [56] . . . 37
3.2 Basic Setup . . . 38
3.3 Mapper . . . 38
3.4 Test Adapter . . . 39
3.5 Model learning setup in LearnLib Studio. Source [56] . . . 40
3.6 Architecture of Tomte . . . 42

4.1 FTP architecture . . . 45
4.2 FTP Control Connection . . . 46
4.3 Heartbeat Messages . . . 54
4.4 Memory leakage . . . 55

5.1 Our basic learning setup . . . 58
5.2 Example of our learning setup for FTP Protocol for the input *Invalid Connect* . . . 67

6.1 Learned Model for Correct Implementation of Heartbeat Protocol . . . 80
6.2 Learned Model for Faulty Implementation of Heartbeat Protocol 80
6.3 Learned Model for Simple FTP Client . . . 86
6.4 Learned Model for FTP4j Client . . . 88
6.5 Learned Model for FTP Client Apache . . . 91
6.6 Example Trace from FTP Client Apache . . . 93

.1 Learned Model from SimpleFTP Client 107
.2 Learned Model from FTP4j Client 108
.3 Learned Model from FTP Client Apache 109

1 Introduction

"The beginning is the most important part of the work [64]. "

[Plato]

The topic of this master thesis is the "Active Learning of Network Protocols" and specifically of the FTP and the Heartbeat protocol. Before describing and analysing the active learning techniques, applied to these protocols, it is important to first understand why we are focusing on network protocols and on these learning techniques and therefore to understand their value. In the next section we present our motivation of this thesis, which is derived from the following factors:

- importance of correctness in software systems
- importance of system's compliance to its specification
- current situation - the problem of implementing correct systems according to their specifications

In addition, after we demonstrate our source of motivation for conducting this thesis, we present the current problem and our approach to the problem: The use of active learning techniques for learning the systems' behavior and revealing possible deviations from their specifications. Lastly, we present some background knowledge on the field of learning theory where active automata learning is based on.

1.1 Motivation

1.1.1 Why correctness of a software system is important and what is the current situation

When building up a software, it is really important to focus on developing it correctly and not to rush its implementation in order to achieve for instance quick results [21]. Correctness is a fundamental aspect that should be taken into account since a correct system is also a robust system that is less prone to errors. A correct system is also a reliable system. Unfortunately, correctness is not always achieved or the developers do not pay a lot of attention on it while implementing a system.

According to [27] more than 75% of implemented software systems are showing errors in the maintenance phase, which increases the software's costs. In addition based on [14], more than 50% of the developing time and effort is spent on testing the system under implementation. A system that lack correctness can lead to errors that may have a catastrophically impact. In 1996 the European Space Agency has launched the Ariane 5 rocket which has been exploded because of a software error in its system. In addition, in 1999 the Mars Climate Orbiter has been lost due to its incorrect implementation; wrong units has been given to the system. As they stated "The problem here was not the error, it was the failure of NASA's systems engineering, and the checks and balances in our processes to detect the error. That's why we lost the spacecraft" [32].

Therefore it is really important to achieve correctness on the system, in other words to check whether it fulfils its requirements and if it is accomplishes what is intended for. The first step for designing a system is to define its specifications; what is expected to deliver, how is expected to behave under any circumstances [59]. A specification can be defined as an agreement between the developers and the stakeholders [59]. In case of software which standardises common functionalities within the Internet, such as protocols, their specification is given by Request for Comments (RFC) [66]. They define how the software should be developed based on what the system should do. In general, a specification can be defined in a formal or an informal way. The former one, it is required a precise syntax and semantics, which

should exactly describe and expose the functionality of the system under development. In the latter case, the system is described in a natural language and is supported by visual notations, tables and diagrams, which help the understanding of the system [59].

Many systems in use today lack adequate specifications or consist of only partial specifications [65]. For that reason, the compliance of the systems to their specifications turns out to be a fundamental issue. A lot of research has been conducted in the field of software testing in order to find a suitable approach for testing the conformance of a system implementation to its formal specifications. Automata learning, a method which has emerged in the 1980's, is an interesting technique that could overcome this situation and anticipate possible errors by learning and modelling the behavior of a system. During the learning phase of an automaton, its behavior is being observed, in particular how the system reacts when generated test sequences are being applied to it [35].

1.1.2 Network Protocols - Incorrect implementations lead to security vulnerabilities

In the previous section we described -as part of developing correct systems- the importance of setting correct specifications to the system under development and to develop the system exactly as specified. In this section we go one step further; we do not refer to whichever software system but we dive into the filed of network protocols. Our motivation of this thesis has been derived specifically from errors and vulnerabilities that have been revealed in two protocols: the File Transfer Protocol (FTP) and the Heartbeat Extension of the OpenSSL Protocol.

Network communication has being used throughout the Internet and forms a basic principle on which the Internet is based on. Protocols which specify rules and conventions for this communication have been developed and used extensively. Hence, protocols that enable reliable communication over a wide network frame an important part of the present computing infrastructure. Network protocols enable the exchange of data between two peers in a way that is understandable by both of them and ensure an effective

communication between them. Due to their importance it is essential to ensure that they are implemented correctly, in other words to ensure that the protocol implementations follow the protocols' standards and comply to their specifications. Possible deviations from the protocols' standards could lead to system vulnerabilities, security flaws, and unexpected behaviour of the system. Therefore, the compliance of the protocol implementation to its specification is a fundamental topic.

Deployment of secure systems is one of the biggest challenges that developers have to face and should always be aware of [34]. For that reason, most security sensitive applications on Internet have adopted a web encryption technique called OpenSSL, which gives a way to secure data and user privacy over the Internet [34]. More specifically, OpenSSL is an open source cryptographic library, which implements the SSL and TLS protocols. SSL stands for Secure Sockets Layer whereby TLS stands for Transport Layer Security. Both protocols provide data encryption and authentication and are used for encrypting the traffic that passes between a web server and a client [62]. It is widely used for securing web connections, email, VPN and other services.

However, over the last thirteen years, attackers have compromised private server data (cryptographic keys, messages in memory etc.) and six code execution vulnerabilities have been documented [18]. On top of that, eight information leak vulnerabilities have occurred, that allowed an attacker to retrieve private keys and/or plaintext. Two of these vulnerabilities have appeared due to protocol weaknesses, where "Heartbeat" is one of them. Heartbeat is an extension of the OpenSSL protocol and is used to check whether the connection between the two peers (server/client) is still "alive". Precondition to this is that the two peers communicate with each other over a TLS connection. This protocol has been introduced in February 2012 by RFC 6520 [79], has been added to the OpenSSL on December, 2011 and released in OpenSSL Version 1.0.1 on March, 2012 [18].

In April 2014, Neel Mehta of Google's security team reported a severe vulnerability in this Heartbeat extension of the OpenSSL, which allows attackers to read confidential encrypted data and also retrieve the encryption keys used to secure this data. Half a million widely trusted websites were vulnerable to this bug and already 17.5% of SSL sites had the heartbeat

extension enabled on. This security flaw is considered to be one of the biggest security threats over the Internet [18], due to the fact that it was easy to be exploited and it was able to reveal private cryptographic keys and private user data.

Moreover, security issues has been encountered also to other network protocols, such as the File Transfer Protocol (FTP). The so called FTP bounce attack [22] enables an attacker to open a connection between an FTP server and a third machine on an arbitrary port. Through this connection an attacker can bypass access control restrictions and scan ports on private networks.

Unfortunately, the number of security incidents will continue to rise until we understand the risks and raise our security awareness, until we focus on developing test methods, verification techniques and formal models, which would lead to an early identification of such crucial security bugs. As a result, on one hand the widely use of these two protocols and on the other hand the security issues that have been encountered on them, lead us to the main focus of this thesis; actively learning the behavior of the FTP and Heartbeat protocol. Abstracted models, which represent the learned behavior of the protocols are afterwards being analyzed in terms of compliance and conformance to the protocol's specifications.

1.2 Our Approach to the problem

The need of correct systems that comply to their specifications is an important aspect when building up a software. For achieving this, lot of research has been conducted in the field of software testing for revealing system's errors, called bugs. But the problem often is that the software systems can be too big or too complex and therefore is essential to initially understand what the system really does, how it behaves, how it reacts when we interact with it. Thus, is fundamental to go one step back and before testing the system to first try to learn it.

This is the approach that we follow in this thesis as an answer to the problems that we described above: learning the system's behavior, try to understand if the implementation of the system corresponds to what has

been initially specified. By applying learning techniques, we can analyse complex systems such as network protocols and by actively learning them we can have also the control over the inputs that we provide to them during the learning.

Much work has been already conducted in learning the behavior of systems, which shows why active automata learning is an appropriate method for proving correctness of systems. Fide Aarts, Joeri de Ruiter et al. [4], have applied learning methods to the software of banking smartcards for evaluating the security of these systems. Unexpected functionality, missing system's behavior, mistakes in the design and the implementation are only a few incidents that can be uncovered by learning techniques. As they stated, by applying the Angluin's L* algorithm to authentic bank cards, they were able to obtain differences between the generated models of the bank cards. The approach of learning network protocols has been also explored from Paul Fiterau-Brostean, Ramon Janssen et al. at their paper [20], where they have explained how the TCP Network protocol can be learned based on observations. They have analyzed different implementations of the protocol in two different operating systems; Windows 8 and Ubuntu. From their analysis they were able to extract that the behavior of the protocol in both platforms is non-compliant to its specification. Our approach for learning protocols is very similar to their work.

Based on the facts presented, we understand the need of learning the behavior of a system. In this thesis we focus on one category of complex systems, the network protocols. Active automata learning can help expose implementations of the protocols standards, which do not follow the rules of the protocols and can result to security vulnerabilities. The criticality of the Heartbeat vulnerability as described above motivated the need of a good solution that could have potentially prevented this bug. For that reason, one of the protocols we have decided to focus on in this work, is the Heartbeat protocol. We analyse if this severe vulnerability of the Heartbeat protocol could be uncovered, by applying automata learning techniques to it. Hence, one of the systems to be learned in this thesis is the Heartbeat extension of the OpenSSL protocol, well-known as the "Heartbleed". Motivated also from the security vulnerability that has occurred in the FTP protocol as also from its widely use, we analyse and apply learning techniques also to FTP protocol.

More specifically, we have used the active learning method, which assumes that there is a teacher who knows the behavior of the system under learn (SUL) and a Learner, who wants to learn the SUL. The Learner asks queries to the teacher and the teacher (which is represented as an Oracle and knows the SUL) answers these queries. By actively asking queries, at the end the Learner learns the behavior of the system and is able to produce a model that represents it.

In order to use this method in practice we need a learning algorithm which is able to communicate and interact with the system. For that reason, we have used the Angluin's algorithm [9], called L*, which is the most well known that is used for active learning. Its characteristic is the use of queries that can be classified into the two categories: (a) Membership queries and (b) Equivalence queries.

Membership queries (Mq): the Learner sends a sequence of input symbols to the teacher and the teacher returns an output-symbols sequence as an answer to these inputs. In other words, for an input symbol a the teacher will respond with an output sequence $Mq(a)$. These queries are used in order to construct a hypothesis model for the SUL.

Equivalence queries (Eq): After the hypothesis has been constructed, the output of the hypothesis will be checked. This output is compared to the output that is produced by the Mealy machine model of the target SUL. If M^h is different from $M^{(SUL)}$, then a counterexample will be produced. On the other hand, if the two models are equivalent, the learned hypothesis M^h is a correct representation of the SUL and therefore the SUL has been learned successfully. These queries are used in order to verify the constructed hypothesis model.

After that, the Learner generates a hypothetical automaton from these equivalence classes [70]. We use Mealy machines - as a subcategory of an automaton - to represent the learned model as also the hypothesis produced during the active learning. Mealy Machines are in detail described in Section 2.

We use the L* algorithm as our learning algorithm, for learning the behavior of the Heartbeat extension of the OpenSSL protocol and the behavior of three client implementations of the FTP protocol. As a learning tool we

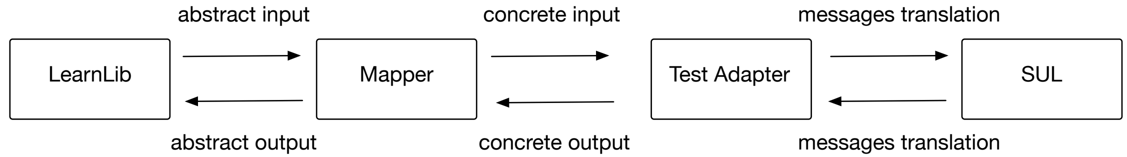

Figure 1.1: Test Adapter

use *LearnLib*, a tool that provides interfaces to learn models of real-life applications. LearnLib is a tool used for active automata learning and sees the system to be learned as a black-box implementation. It is the basic tool that we use in this thesis and is able to learn models which consist of up to thousands of states (Chapter 3.1). LearnLib provides already the implementation of the L* algorithm, so the Learner components is already implemented. What we have done is to connect the three implementations of the FTP with the Learner (in our case LearnLib) and represent them as a Mealy Machine, which can receive inputs, produce outputs and based on the output move to a next state. Regarding the Heartbeat protocol, since it would have been a big effort to connect the real Heartbeat implementation with our Learner, we implemented our own Heartbeat protocol based on the official protocol specification.

Thus, for this master thesis we have used/implemented the following components for performing active automata learning, as also illustrated in Figure 3.4

Learner:

We have used the L* algorithm, which is already implemented in LearnLib.

SUL:

We have set under learn three implementations of FTP, the Heartbeat protocol correct implementation (implemented by us), Heartbeat protocol faulty implementation (implemented by us)

Our purpose for the three FTP implementations:

Our purpose for FTP is to learn, for simplicity reasons, a part of the FTP functionality and not the whole functionality that the FTP offers. Thus, we query the following functions of the SULs: we perform a *connect*, a *login*, a *change of directory*, an *upload* and a *disconnect*. The Learner queries these functions to the SUL in an arbitrary order, which is the way how dependencies are created. For example, you can not perform a *login* if a *connect* has not occurred before. The SUL will produce a certain output if you send a *connect* input and then a *login* input to it, which should be different to the output produced if you sent a *login* input first and a *connect* input afterwards. This is the actually the logic how the behavior of each FTP implementation will be at the end learned.

All the FTP Clients under learn have the same learning setup. The three different FTP client implementations that we analyse are:

- SimpleFTP
- FTP from Apache
- FTP4j Client

SimpleFTP [3] is a very simple FTP Client, used for performing simple actions with the FTP Server. On the contrary, the FTPClient provided by Apache [26], is the most well-known and used FTPClient that incorporates the whole functionality of the FTP protocol. It performs all possible actions that could be accomplished from a client to an FTP Server. Last but not least, the FTP4jClient [25] is a library that also implements in Java the full-features of an FTP Client and supports different security authentications. These three implementations have been chosen due to their different complexity and their different use percentage, in order to examine how these various clients behave in respect to the specification of the FTP protocol as described at RFC [19].

Our purpose for Heartbeat protocol:

Regarding the Heartbeat protocol, our purpose is to learn the behavior of the correct implementation as also the behavior of the faulty one. By comparing these two learned models, we can observe any unexpected behavior of the protocol and uncover serious issues, as the Heartbleed vulnerability. Since the real implementation of the Heartbeat protocol was not so trivial to set it under learn, we have implemented our own implementations; one that complies to the specification and one with the Heartbleed vulnerability in it.

Mapper:

Protocols are defined as more complex systems that are using a lot of parameters and variables for connection establishment and synchronisation. For that reason, these numerous parameters could be reduced, by using abstraction techniques. More specifically, in [35] was illustrated the use of a mapper that is placed between the Learner and the system under learn. This mapper abstracts the values of the parameters and the variables used in the protocol, creates a smaller set and therefore learning algorithms can be more effectively applied. The mapper abstracts the concrete outputs, which are sent from the SUL and sends the abstracted outputs as inputs to the Learner. On the other side, the Learner sends abstract input symbols to the mapper, which maps them to concrete ones and sends them as input to the SUL. When a reset message is sent from the Learner to the SUL, a reset message is also sent the same moment to the mapper component. The basic element of the mapper is the abstraction function. As stated in [5], a mapper is also considered to be a Mealy machine, which has as inputs and outputs the concrete and abstract sets, with a set of mapper states, with an initial mapper state with a transition function and also an abstraction function.

For our work, we have also used abstraction, based on the approach [20] for learning the implementations of the two network protocols, by implementing our own mapper component and placing it between the Learner and the SUL. More details about the mapper construction and the abstraction of the input/outputs that we have performed can be found on section 5.2.2 for FTP and on section 5.1, for Heartbeat protocol.

Test Adapter:

As described above, the mapper is responsible for mapping abstract and concrete values between LearnLib and the SUL. However, the concrete values that are produced by the mapper have to be passed to the SUL in a way that it understands them. Therefore, they have to be transformed in the right format that can be read by the SUL. For that reason, another component that is called *test adapter* has to be placed between the mapper and the SUL. As shown in Figure 3.4 the test adapter receives the concrete values from the mapper and translates them to an understandable format for the SUL. The mapper and the test adapter components are not per default implemented from the LearnLib library; they are depending each time on the SUL's methods and therefore we have implement them manually. In Section 5 the practical aspects of the mapper and test adapter components, which have been created for the learning process, are in detail described.

Results

Our results have shown that the FTP client implementations were able to be learned for the two out of the three users that we have created for authentication to the FTP server. Due to a non-deterministic behavior of the server, no model was able to be constructed for the third user, because the learning algorithm terminated with an exception. When the *user3* tries to be authenticated, the FTP server sometimes resets the connection and some other times not. This results to different outputs and state transitions for the same inputs and leads to non-determinism. Furthermore, we demonstrate occasions where these FTP implementations do not adhere to the standards and they appear unexpected behavior. In particular, by using learning techniques we were able to discover a misbehavior of the implementation of the FTP client of Apache, where we reported the issue to the Apache developers. Moreover, we found out that the FTP4j client does not comply with the FTP protocol's specification as defined in RFC [66]. The SimpleFTP was the only protocol that complied to the specifications and did not show any unexpected behavior.

Regarding the Heartbeat protocol, our results have shown that by using the learning setup of this work, we were not able to detect the Heartbleed vulnerability as expected. The learning models for the correct and for

the faulty implementation have resulted to models with just one state. The resulted model has shown that different inputs will produce different outputs, but will result to the same state. That is due to the fact, that every message sent is an independent event from the next message that will be sent. Although the learning was not as we expected, we could state that the comparison of these one-state models (as shown in 6.1, 6.2), could reveal that for the same input a different output is generated. As a result, since the models are not identical, this could be a basis for a further investigation of this observation.

As we concluded in section 6, where our results are presented, one of the most important actions to be taken into account when applying learning techniques, is the the correct setup and implementation of the mapper component. A not so precise implementation of the mapper, can result in divergent final results, as we figured out throughout this thesis. Even a small change to the mapper behavior can lead to a dissimilar outcome.

Moreover, we analyse in our work and compare statistical information and benchmarks of the learned models, such as execution time, number of membership and equivalence queries, number of states etc. by using two different algorithms for equivalence verification applied in the three FTP client implementations. The equivalence algorithms used are the 'Random Walks' and the 'W-method' and will be analytically presented in chapter 3. Based on our results, by using the W-method the time of equivalence queries has been extremely reduced although the overall execution time has not been much affected.

Since the application of active automata learning on the two previously described network protocols will be the main focus of this thesis, a brief introduction to the Learning Theory will be presented in the next section.

1.3 Background knowledge of the Learning Theory

As described above, we use active learning techniques for learning the behavior of a system and then evaluating if the learned behavior is the

expected one as initially specified. Active learning is one specific method of the "Learning Theory" field that can be used for proving the correctness of the system. This method will be analytically described in the next chapter (Chapter 2), since it is the basic approach used in this thesis. However, the idea of "Learning a system" includes also other learning methods and can be also applied in different scientific fields. Therefore, in this section a background of what is meant by "learning" the behavior of the system will be given, as also a description of the basic aspects of the so called "Learning Theory" -the field where Active Learning also comes from.

1.3.1 Machine Learning & the three basic types of Learning

In general, learning is defined as a "relatively permanent change" in the behavior of a system, which is derived from the past experience and a learning system is described by its ability to improve its behavior during the time by converging to a required objective [58]. In the context of designing machines that are able to learn, learning can also denote the inference of rules derived from examples [67]. Thus, machine learning deals with the problem of automatically generating system descriptions [65] and creating algorithms that can learn from the data they process and analyse [38]. In other words, the general idea of machine learning can be summarised by the following actions: first collecting the data and afterwards using this data in order to learn how to solve a problem or a task. As a result, the learning problems can be formulated as a relationship (mapping) between inputs and outputs. What we try to learn is, which is the best produced output for each possible input [8]. For instance, in case of face recognition, the input would be an image and the output the identified faces, in speech recognition the input would be a speech signal and the output a text sentence. Machine learning can be used in several scientific fields such as artificial intelligence, theoretical computer science, pattern recognition and more [69]. Various computation structures can be learned such as logic programs, rules, grammars as also finite state automata.

In machine learning theory we can choose which learning method to use based on the kind of data that we are dealing with. In particular, the examples of data can either have both the inputs and the outputs *(supervised*

learning) or only the inputs (unsupervised learning). In other cases, we may can not access directly the correct output but we can estimate the quality of an output o following the input i (reinforcement learning). In the former case when these example of data, called training data, are encoded as pairs of inputs-outputs (i, o) we refer to this type of learning as *supervised learning*. The learning function depends often on a set of parameters and the learning goal is defined by adjusting these parameters in order to fit the data. As a good learning model is a model that generalises well to new data [8], which means that is able to abstract apart for detecting underlying patterns. On the other hand, on unsupervised learning where only the collection of input examples is available, we try to learn the patterns of the data by applying clustering techniques based on similarities or correlations between features of the data. Lastly, on reinforcement learning although there is no direct access to the correct output for an input, we have a measure of how good or bad an output is. In that way, we can learn the behavior that maximises the expected goal by observations. A so called 'agent' executes a sequence of actions to the environment and the environment is providing a signal (reward) as a feedback to these actions. This numerical signal reward tries the agent to maximise. A Learner relates this reward signal to previously executed actions in order to learn a "policy" which will maximise the expected future reward [61]. This way of learning is also used in cognitive science and in behavioural psychology.

1.3.2 From Machine Learning to Automata Learning

Machine Learning is one important area of the Learning Theory that we described above. Another field of Learning is the " Automata Learning", which focuses on solving computational problems by making use of these automata. As stated in [61] *Learning Automata are adaptive decision making devices suited for operation in unknown stochastic environments.* They have first been introduced in mathematical psychology and have been used to describe the human behavior from a psychological and biological view [61]. However, they have been used nowadays mostly in the field of engineering. A variable structure of a learning automaton is very close to the concept of the reinforcement learning -as described above- and more specifically are close

associated to a Reinforcement Learner of a policy iteration type. In particular, the learning automaton at each time has a probability $p(t) = (p1(t).., pl(t))$, where $p1(t)$ is the probability for selecting the first action at time t and l stands for the number of actions. After an action has been performed by the automaton, the environment produces a reinforcement signal. These probabilities are being adjusted over time based on this produced signal. Therefore, learning automata can be defined as policy iterators, since playing the role of reinforcement Learners, they update the policies when an action i is selected and a reward $r(t)$ is received at time t. Learning automata have been used to a wide range of modelling and control problems, due to their fast and accurate converge with low computational complexity [61].

In this thesis we focus on a specific case of Automata Learning called Active Automata Learning, that will be presented in detail in Chapter 2 "Preliminaries"".

1.4 Overview and organisation of this thesis

This thesis will address the following questions about applying Automata Learning Techniques to a system under learn (SUL):

- Could possible errors and unexpected system's behavior be uncovered through applied automata learning techniques?
- How big is the complexity and how high is the effort of learning a system?
- Do the network protocols that are being learned comply to their specifications?
- How is the performance of the "learning" affected when different equivalence methods are used and which optimisations can we perform?

Based on our work and our results we are able to answer the above questions, which we have set as our main objectives for this thesis. We present our answers to these question at the "Experimental Results and Discussion" Section (Chapter 6). The organisation of this thesis is as follows:

The first part of the thesis (Chapters 2 to 3) provides a basic knowledge background related to the thesis focus and purpose. Initially, Chapter 2 sets the basic concepts for state machines and formal languages, which are used in learning algorithms. The main focus of this thesis concerns automata learning methods. Therefore, in Chapter 2.1.1, we present a brief description of the automata learning algorithm used in this work. Chapter 3 gives an overview of some of the tools, which are used for active automata learning. The tools that we have used in this master thesis, such as LearnLib, are presented in a more detail way. In Chapter 4 we describe the network protocols under learn as also their formal specifications.

The second part of the thesis (Chapter 5) describes the technical implementation and shows the experiments that have been conducted. In particular, we present extendedly the learning setup and the description of the learning components.

Finally, in the third part of the thesis we expose our results, conclusions and future work. More precisely, in Chapter 6 we discuss the current results and we evaluate the final outcome of our learning process. Additionally, in Chapter 7 we present some interesting relevant papers and related to this master thesis work. Lastly, in Chapter 8 we give answers to all of the goals that have been set at the beginning of this thesis and we outline some ideas for future work and further research.

2 Preliminaries and Basic Concepts of Automata Learning

> *"There comes a time when the mind takes a higher plane of knowledge but can never prove how it got there [39]. "*
>
> [Albert Einstein]

Automaton Theory is the field of theoretical computer science which studies abstracted machines called "automaton" and focuses on solving computational problems by making use of these automata.

An automaton has been initially used to represent a finite description of a formal language. Thus, a given formal language can be recognised by an automaton which describes it and accepts it. A formal language is a set of strings, which are defined over an alphabet and are accepted by an automaton representation [31]. Let's assume that the alphabet of a language is $\Sigma = a, b$ and is accepted by the automaton M. This language contains all input strings accepted by M. By giving input strings to a (finite) automaton, the automaton will transit to a state. This end state will be either a "rejected" or an "accepted" state, depending on where the given input string belongs to the alphabet of the language or not. In the diagram 2.1 we represent a simple automaton which given an input 0 or 1 it can transit to a next state. The state e, which has a double line, represents the accepting state of this automaton. The "language" that this automaton accepts, allows an infinite number of words.

Therefore, an automaton consists of [31]:

1. a set of states
2. a starting state
3. a set of accepting states

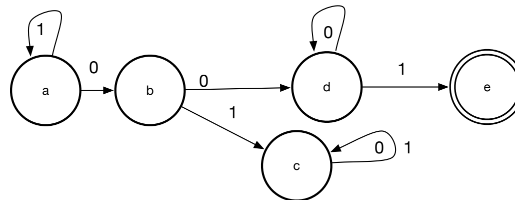

Figure 2.1: Example of an automaton with state e as accepting state

4. a set of transitions from one state to another
5. a set of inputs to the automaton

In this section we shortly described a general automaton definition. In the next sections, a more specific usage of the automaton will be introduced; the Finite State Machines as also the Mealy Machines.

Finite State Machines

One of the most widely used representations of automaton is the Finite-state machine (FSM). As the name itself indicates, a Finite State Machine is an automaton with a finite number of states. Therefore, it consists of the same elements as an automaton; input, output, initial state, finite number of states and a transition function. The transition functions is a function which maps every given input to a certain output and at the same time transits to a state [31].

A finite-state machine is formally defined as a five tuple [28]:

$$M = \langle Q, I, Z, \theta, q0 \rangle \tag{2.1}$$

where:

- Q = finite set of states

- I = finite set of input symbols
- Z = finite set of output symbols
- θ = mapping of I $\times$ Q into Q called the state transition function, i.e. I $\times$ Q $\rightarrow$ Q
- qo $\in$ Q = initial state

Mealy Machines

In this work, we model our systems as finite state machines and more specifically as Mealy machines. A Mealy machine is special form of a finite state machine, which forms a well-known category of automaton. The outputs that a Mealy machine produces are a function of the given input and of the state. Thus, the output is dependent on the current state and input. We described above an automaton as a system which by receiving an input produce an output and will move to an "accepted" or "rejected" state. However, Mealy machines do not distinguish between accepting and rejecting states but their aim is to produce a certain output based on their input.

Mealy machines are a way to model reactive systems formally. A Mealy machine is a variant of automata which is defined as a tuple [4]

$$M = \langle S, S0, \Sigma, \Omega, \delta, \lambda \rangle, \quad (2.2)$$

where:

- S is a finite nonempty set of states (where n = $|S|$ the size of the Mealy machine),
- So $\in$ S is the initial state,
- Σ is a finite input alphabet,
- Ω is a finite output alphabet,
- $\delta : S \times \Sigma \rightarrow S$ is the transition function, and
- $\lambda : S \times \Sigma \rightarrow \Omega$ is the output function.

The machine can be at any state $s \in S$. From this state it is possible to give an input to the machine $\alpha \in \Sigma$ and it will respond by producing an output symbol $\lambda(s, \alpha)$. At the same time it will transform itself to a new state $\delta(s, \alpha)$.

For instance, the transition $s \rightarrow s'$ means that when an input symbol $\alpha \in \Sigma$ is received, the Mealy machine moves from state s to state s′ and produces the output symbol $o \in \Omega$. Elements Σ and Ω are input and output alphabets respectively.

In our work, we use Mealy machines in order to model the behavior of the system under learn. We use input and output strings instead of symbols and we consider in this thesis that the Mealy machines are complete and deterministic. Therefore, for each state s and input Σ the automaton can move only to one next state $\delta(s, \alpha)$ and will produce only one output $\lambda(s, \alpha)$.

Mealy Machines are moving through states. In order to minimise the produced states of a Mealy Machine to a smaller set, it is needed to identify if equivalent states are produced. If so, then they can be represented by one state. Moreover, equivalence is also needed when comparing if two produced Mealy Machines are equal. Therefore, below we present definitions as described in [56], for a better understanding on what is meant by "Equivalence between classes" on Mealy Machines.

For Mealy machines equivalence between states is defined as [56]:

Two states $s\alpha$, $sb \in S$ of a Mealy machine are equivalent if both states show exactly the same behavior for all futures, i.e., every input word $w \in \Sigma+$ produces the same output for both:

$$s\alpha \equiv sb \iff \lambda(s\alpha, w) = \lambda(sb, w) \forall w \in \Sigma+ \tag{2.3}$$

The inputs of a Mealy Machine can be words by means of a set of strings. Based on the words which are given as input, a "next state" will be defined. Hence, the predefined state equivalence can be exposed by using the so called "Nerode relation" on languages (set of words), which is defined as follows [56]:

Two words $x, y \in \Sigma$ are considered equivalent by the Nerode relation on the language L if for any suffix $z \in \Sigma^*$ the concatenations xz and yz are either members or non-members of language

$$L\colon x \sim Ly \iff (\forall z \in \Sigma^* : xz \in L \iff yz \in L) \tag{2.4}$$

Hence, the equivalence between classes can be also defined as follows [56]:

For $x \in \Sigma^*$, the equivalence class [x] is defined as

$$[x] = \{y \in \Sigma^* \mid x \sim Ly\} \tag{2.5}$$

In other words, if two states produce the same output behavior for all the future events, these states are equivalent, based on the above Mealy machine specification for state equivalence. Thus, these states can be combined to one state and a minimal Mealy machine automaton will be produced.

2.1 Passive and Active Automata learning

Automata learning can be divided into two learning categories: passive and active learning. Passive learning algorithms are based on a given set of observations, which has been already pre-collected, in order to learn a model and do not actively ask for additional observations and information. In passive learning the learning process can be divided into three steps [50]. Initially, the system under learn in observed and its behavior is being stored for later analysis. After that, the so called "prefix tree acceptor" is created, which is a deterministic tree graph, where given a set of strings (words), it will create a next state for each of these strings. If the state already exists, no new state will be created. For example for the set of strings {baa, baba} the prefix tree acceptor as shown in Figure 2.2 will be created. The creation of this tree is based on the previous stored observations.

Lastly, the states that are compatible with the input will be represented only by one generalised state.

The main issue of this approach is the fact that the construction of an accurate model that represents the system's behavior is not so trivial. The prerecorded data may not be enough and may not give all the information required about the system under learn. The problem lays to the fact that we are learning the system "offline" and therefore we can not ask for more information and data at any time we would need them. A good solution to that would be to request more information about the system any time

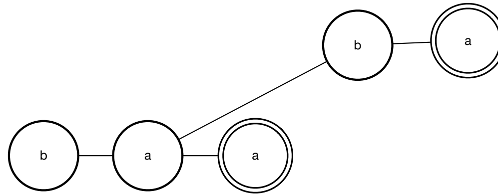

Figure 2.2: The prefix tree acceptor for {baa, baba}

there is an ambiguity and a "decision" has to be made. This approach is introduced by active automata learning and will be presented in the following section.

The problem stated with passive automata learning can be solved by using another learning approach; the active learning. In the active learning approach, it is assumed that there is a teacher who knows the behavior of the system under learn (SUL) and a learner, who wants to learn the SUL. The learner asks queries to the teacher and the teacher (which is represented as an Oracle and knows the SUL) answers these queries. The difference to the passive learning approach is the fact that the learning procedure is "active", so at any time the learner needs more information about the SUL, the teacher will be queried and therefore ambiguities can be avoided and more knowledge about the SUL is gained. The basic representation of active learning is the well-known algorithm of Angluin, called L*. In this algorithm the concept of the *minimally adequate teacher (MAT)* model has been introduced, that is presented below.

In terms of convenience we will use the same symbols defined in section "Mealy machine" to describe the initial state, the finite input alphabet, finite output alphabet, transition function, and output function.

The MAT model assumes there is a *Teacher* which knows the system's behavior, which is represented as a deterministic Mealy machine [56]

$$M = \langle S, s0, \Sigma, \Omega, \delta, \lambda \rangle \tag{2.6}$$

Thus, the teacher is able to answer two types of queries [56]:

- Membership queries (Mq): the learner sends a sequence of input symbols to the teacher and the teacher returns an output-symbols sequence as an answer to these inputs. In other words, for an input symbol a the teacher will response with an output sequence *Mq(a)*. These queries are used in order to construct a hypothesis model for the SUL.
- Equivalence queries (Eq): After the hypothesis has been constructed

$$M^h = \langle S^h, s0^h, \Sigma^h, \Omega^h, \delta^h, \lambda^h \rangle \tag{2.7}$$

, the output of the hypothesis will be checked. This output is compared to the output that is produced by the Mealy machine model of the target SUL.

$$M^{(SUL)} = \langle S^{(SUL)}, s0^{(SUL)}, \Sigma^{(SUL)}, \Omega^{(SUL)}, \delta^{(SUL)}, \lambda^{(SUL)} \rangle \tag{2.8}$$

If M^h is different from $M^{(SUL)}$, then a counterexample will be produced. On the other hand, if the two models are equivalent, the learned hypothesis M^h is a correct representation of the SUL and therefore the SUL has been learned successfully. These queries are used in order to verify the constructed hypothesis model.

This is a loop process, where starting from the Mq phase and proceeding with the Eq phase, the loop will stop either when no counterexample is produced or after some predefined number of steps. In the former case, a correct learned model will be defined. In the next section the well known Angluin's L* algorithm, which has first introduced and implemented the MAT model will be presented.

2.1.1 Angluin's L* Algorithm

Angluin's algorithm [9], called L* is the most well known one that is used to infer deterministic finite automata, like Mealy machines. Its characteristic is

the use of queries that can be classified into the two categories as described above: (a) Membership queries and (b) Equivalence queries. The purpose of Angluin's L* is to observe the SUL's behavior by gathering enough knowledge about it and formulate a hypothetical model of it. It is a learning system which can improve its behavior during the running time and aims to provide behavioral equivalence between the learned automaton and the target automaton. Thus, the learner actively reacts with the automaton output to identify equivalence classes in the input sequences. After that, the learner generates a hypothetical automaton from these equivalence classes [70].

A representation of the L* algorithm is presented in Figure 2.3, where the L* algorithm takes the role of the "learner". As described by Angluin [9], Chen [15] and Berg [12] et al. this algorithm consists of several components, which interact with each other throughout the whole learning procedure. These components are:

- Learner: It is the core component that controls the learning process by querying the teacher. Learner can be any learning algorithm, in this case, it is the L* algorithm.
- Teacher: It responses to the informational needs of the learner with data from the SUL.
- SUL: The system under learn that is represented as an deterministic automaton (in this case as Mealy machine).
- Observation Table: It consists of all the information that the learner knows about the SUL and is updated at each run. At the end, a behaviorally equivalent system to the SUL is retrieved based on this observed data.

Since the SUL is considered to be a black-box system, the only way that the learner can retrieve information from the system is by creating queries and sending them to the teacher. The teacher, that knows the internal structure of the SUL, answers these queries by producing an output to each input query. The teacher acts as an oracle since it needs to know the representation of the SUL. The use of queries gives information for the way the system reacts to the given inputs and finally the system's behavior is learned. As in every active learning procedure two roles are determined as already described above: the learner, who asks questions to the system and the teacher, who

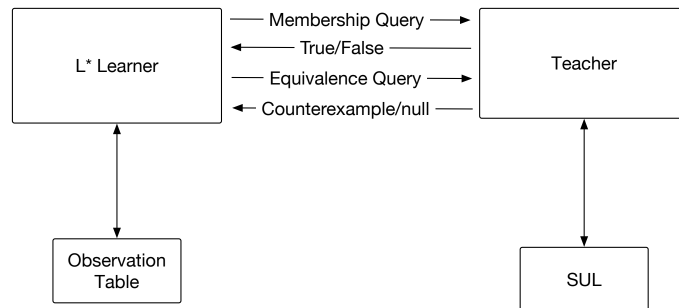

Figure 2.3: Components of the L* Algorithm

knows the deterministic Mealy machine M = $\langle S, s0, \Sigma, \Omega, \delta, \lambda \rangle$. The learner initially has only knowledge for the input and output alphabet that can "use" in order to query the teacher-Mealy machine. Hence, it identifies the accepted language by the automaton to be learned. The learning procedure is a simple mapping of words from the input alphabet that the learner provides to true, false depending on wether the automaton accepts or rejects the input words [70]. "Accept" means that the input received, leads to an accepting state and "reject" to a rejecting one, according to the transition function.

The L* algorithm in detail

In this section, the L* algorithm is presented from a more detailed view. The L* algorithm aims to learn an unknown regular language L and construct deterministic finite state machine (in our case a Mealy machine) that accepts L. Based on the MAT model described before, by making use of membership

queries tries to find out if a certain string $s \in \Sigma^*$, where $Sigma^*$ is the alphabet, belongs to the language L. When asking membership queries to the teacher, at the same time it fills in the so called *observation table* with the provided information. After that, based on this information from the observation table, the algorithm calculates and constructs a Mealy machine hypothesis. We define the observation table (S, P, T) as a set of suffixes S, prefixes P , both respectively over alphabet Σ^* and transition function T. The transition function is mapping $(S \cup S \cdot \Sigma) \cdot P$ to $\{true, false\}$ and the empty string is represented by ϵ. The pseudocode the L* algorithm can be represented, as stated in [63], as follows:

```
1. Let S= P= {λ}
loop{
2. Update transition function T using queries
   While (S,P,T) is not closed{
3. Add sa to S, so that S becomes closed, where s ∈ S and a ∈ Σ
4. Update T using queries
   }
5. Construct candidate Mealy machine automaton M^h from (S,P,T)
6. Make the conjecture of M^h
7. If M^h is correct
    then return the M^h
   else
     add s (suffix) ∈ Σ* that witnesses the counterexample to P

}
```

The termination of the algorithm occurs when *consistency and closeness* of the observation table is achieved.

Consistency is defined if from one state you can transit to exactly to one successor state. If not, then the observation table is non-consistent and its behavior in non-deterministic [56].

Closeness is defined when all transitions lead to states that already exist [56]. In particular, for all the new states their successors already exist (so the state has already been established), then the observation table is "closed".

Based on this information, that the table contains, the algorithm constructs a hypothesis as a deterministic conjecture and asks the teacher for each conjecture, whether its is correct or not. In the former case, the algorithm will terminate. In the latter case, the L* will use the response of the teacher as a counterexample in order to extend the observation table with strings which indicate differences between the target SUL (unknown language L) and the hypothesis automaton [63].

L* algorithm terminates with a minimal automaton M which represents the unknown language L. Thus, it guarantees that any other Mealy automaton that accepts the language L will have at least as many states as M^h and maybe more [63]. The L* algorithm requires only a polynomial number of queries in order to find a hypothesis M^h such that $M^h \sim L$, whereby the learning of the smallest deterministic automaton is well-known to be NP-hard [6].

2.2 Black Box System

In this master thesis, we look at the network protocols' implementations that we analyse as black-box systems. Black box is considered a system, where no knowledge of the programming language that has been used is needed, no information on the implementation of the system is required and no a-priori insight in their code. In order to test such systems black-box testing techniques can be applied such as[59]:

- State Transition Diagrams or State Graphs
- Equivalence Class Partitioning
- Decision Tables
- All Pairs Technique
- Boundary Value Analysis
- Orthogonal Arrays

We will not describe and analyse the other techniques, but only the state graphs, since they are relevant to this thesis because we model our learned systems as state machines. State graphs are an exceptional way for exposing the system's requirements and modelling its behavior [59]. The system can

be modelled as a finite state machine. By starting from the start state, the system responses to the given inputs with a produced output. When the system responses to the same input with a different output, then it transits to another state than the current one. If the produced output is the same, then the system stays at the same state. In that way, the whole system's behavior and reaction to inputs can be summarised as a state diagram, which consists of states and in-between transitions [69]. After that, various test cases can be derived from this graph, since each node can be defined as an individual test case. By comparing the observed values with the expected ones, the test cases pass or fail. Hence, the main purpose of this kind of testing is revealing inconsistencies between the system's behavior and the predefined system's requirements. Although in this master thesis we will not perform black-box testing methods, we look however at the input/output sets that are passed and produced to/from the system under learn (SUL). The implementation code of the network protocols are not known to us and therefore we handle them as black-box systems.

3 Active Learning Tools

> *"No amount of experimentation can ever prove me right; a single experiment can prove me wrong [85]. "*
>
> [Albert Einstein]

In this section we present some tools, which can be used for actively learning the behavior of systems. Initially, the main tool that has been used in this thesis for active learning some implementations of network protocols, LearnLib, will be presented. Moreover, LearnLib Studio, a modelling and execution environment for learning setups as also Tomte, another tool which can be used when applying automata learning techniques will be shortly demonstrated. However, the focus of this thesis will be on LearnLib, a java library of tools for automata learning [57].

3.1 LearnLib

LearnLib is a tool used for active automata learning and sees the system to be learned as a black-box implementation, as described in Section 2.2. It is the basic tool that we use in this thesis and is able to learn models which consist of up to thousands of states. It is able to learn Mealy machines or DFA's using the L* algorithm as also other learning algorithms. Moreover, it contains lots of optimisation techniques, such as optimising and eliminating the number of queries asked by the corresponding algorithm. In addition, it implements various learning strategies and uses different ways to generate membership queries and finally constructs a behavioural model [48]. Therefore, LearnLib tool is basically a framework that can be effectively used for automata learning. More specifically, it consists of a Java library

that includes a Java implementation of learning algorithms, as also the L* algorithm that has been analytically presented in Section 2.1.1.

LearLib has been initially used for building up finite state machine models generated from real systems [65]. However, this initial intention has been changed and nowadays LearnLib is used as a platform which provides different learning algorithms that are not only used by the users but also extended and optimised. An example of such optimisations is the elimination of the size of membership queries [48].

The main libraries that LearnLib consists of are [65]. :

- Automata Learning framework, where the learning algorithms are provided
- Filter library, which reduces the number of queries, through optimisation methods
- Approximative equivalence queries library, which is used for generation of conformance test suites for the construction of the hypothesis

LearnLib focuses basically on three main categories: on learning algorithms, on the so called "infrastructure components" and on methods for finding counterexamples[40].

The first category that LearnLib is focusing on is the learning algorithms. The basic learning algorithm of LearnLib is the Angluin's algorithm, which is also the algorithm used in this thesis. As described in Section 2.1.1, Angluin's teacher oracle can answer two kind of queries: Membership queries in terms of word sets, which are accepted or not accepted from the finite state machine and Equivalence queries, in terms of whether the constructed hypothesis corresponds to the finite state machine that is examined. The first type of queries can be answered directly from the system under learning whereby the Equivalence queries can be answered by a "Yes" or by producing a "Counterexample" [65]. However, other algorithms such as the *Observation Pack algorithm*, *Kearns & Vazirani's algorithm* as also the *DHC algorithm* are also available in LearnLib. All of them, except the DHC, can be used for learning Mealy and DFA automata [40].

The second category where LearnLib is targeting on is on providing an infrastructure library for optimising the learning procedure. For example, it

uses cash filters which "remember" that this query has been generated and used before and therefore it retrieves and uses this information from the cash. Therefore, LearnLib does not need to create another duplicate-query and this results to faster results. In addition, filters that are used for reducing the number of "reset" actions or other statistics that help to speed up the learning procedure, are part of the infrastructure components that LearnLib provides [**c**].

The last category that LearnLib is focusing on, is the construction of Equivalence queries. In case a model of the target system is available, LearnLib is using the *Hopcroft and Karp's* [71] algorithm for equivalence verification. In case the system under test is a black-box implementation, approximation methods are used for generating the equivalence queries. LearnLib uses then conformance testing for building equivalence queries and finding counterexamples. More specifically, LearnLib finds counterexamples by using the following methods [49]:

- Randomised Depth-First Search (Random Walk only for Mealy machines)
- Random Test cases (Random Words)
- Exhaustive generation of test inputs (Breadth-First Search up to a certain depth k)
- Conformance Testing (W-method and Wp-Method)

For this master thesis, the Randomised DFS and the W-method will be used and therefore they are explained in the next sections.

Random Walks

When using conformance testing, missing states can be also identified. One way to do that is by using randomised DFS. DFS stands for the Depth-First Search algorithm [37] which, given an initial state *i* of the graph, examines whether every other state can be reached from this state *i*. Thus, randomized DFS is starting from a random state in the graph and recursively searches any other state that is reachable from this state in one execution step. After that, from this next state it searches if any other possible state can be reached. This procedure continues as soon as no other state is reachable from the

previous state [37]. The algorithm terminates when all the states of the graph are visited. Since the hypothesis in Learnlib is described as a Mealy machine graph, this algorithm can be used in order to check weather the hypothesis-graph is equivalent to the Mealy machine graph of the SUL. If not, then a counterexample is produced and the hypothesis is refined.

W-method

The second method that we have used in LearnLib for founding counterexamples, is the W-method. The W-method has been first proposed by Chow [44] and is used for constructing test cases derived from a given finite state machine (FSM). The generated test cases are a finite set of sequences, which can be afterwards passed to an FSM in order to test its correctness [52] when its specification is known. Therefore, LearnLib uses the W-method to generate test sequences (equivalence queries), which are then used for generation of conformance test suites for the construction of the hypothesis. As described in [52] for a given Mealy machine

$$M = \langle S, s0, \Sigma, \Omega, \delta, \lambda \rangle \tag{3.1}$$

(for our purposes we use Mealy machine as an FSM), the W-method will perform the following steps, for generating tests from M:

- Step1: estimation of the maximum states in the correct design
- Step2: generation of the so called characterisation "W set" for the given Mealy machine M
- Step3: construction of the testing tree for M and definition of the transition cover set P
- Step4: construction of the set Z
- Step5: the product P x Z is the final test set

These steps will be analytically described in this section. Starting from Step1, since we have no access to the correct design model, an estimation of the maximum states should be performed [52]. Therefore, if no information of the correct implementation is available, in the worst case scenario the

maximum states would be the number of states in M. In Step2, the characterisation W-set will be derived from M. A characterisation set W is defined as follows [52]:

Definition: A characterisation set W for a minimal and complete Mealy machine M, is a finite set of input sequences which distinguish the behavior of any pair of states in M. Hence, for example for two states s_0, s_1 *in S, if there is a string a* $\in \Sigma^*$ *such that* $\lambda(s_0, a) \neq \lambda(s_1, a)$ *, then this string belongs to the characterisation "W-Set"*

In other words, if we start from the state s_0 and we apply this string a, then an output a' will be produced. But if starting from state s_1 and we apply the same string a, then another output a'' will be produced, were $a' \neq a''$. Therefore, a belongs to the set W.

In Step3, the construction of a transition cover set is defined. A transition cover set P as described in [52], can be constructed by using a so called *testing tree* of M. Thus, the first action at this step would be to construct the testing tree of M.

Definition [52]: Let us assume that s_1, s_2*, where* $s_1 \neq s_2$*, are two states in M. The transition cover set* P *consists of the empty sequence* ϵ *and the sequences a,b such that* $\delta(s_0, a) = s_1$ *and* $delta(s_1, b) = s_2$

Therefore, starting from the initial state s_0, we apply all the possible set of strings at this state. By applying each string, we check if the next state that M moves on is again the state s_0. If this is the case then s_0 becomes a leaf. If the next state that M moves on is another state, for instance state s_3, different to s_0, then we continue the same procedure starting from s_3, since now s_3 is the "root". In general, a leaf node means that this state has already occurred before (has appeared at least once as a state). At the end, we have constructed a tree, where all the states of M are present. After the construction of this Tree, we start from the root (in our case s_0) and we go through each path that ends to a leaf node. The empty string together with the set of all these paths form the transition cover set P. With this method we are sure that all the states of M are reached.

At Step 4, the construction of the Z set takes place. The set Z is defined as follows [52]:

$Z = (\Sigma^0 \cdot W) \cup (\Sigma \cdot W) \cup (\Sigma^2 \cdot W) \cup ...(\Sigma^{(m-1-n)} \cdot W) \cdot (\Sigma^{(m-n)} \cdot W)$

where Σ is the input alphabet and W is the characterisation set as described in Step2. The number of maximum states which has been estimated in Step1 in the correct design is n whereby m stands for the number of states in the M, with $m > n$. As a result, in case of $m = n$ then $Z = W$. The symbol $(\cdot)$ stands for string concatenation. In case of $m < n$ then $Z = W\Sigma$.

Lastly, in the Step5 we derive a test set T by concatenating Z with P. In Step 3, we have calculated the transition cover set P, where in Step 4 we have constructed the set Z. The product of these two sets is our final test set. This test set will be given as input to the SUL in order to test if the two models (the SUL and the hypothesis model) are equivalent. LearnLib is using this method for approximating the equivalence queries. Thus, the learning algorithm will apply each input t from the set $P \cdot Z$ and compare the hypothesis *M(t)* with the implementation *SUL(t)*. In case they are equal the hypothesis is correct otherwise a counterexample is produced and the hypothesis has to be refined.

Wp-method

Another method that LearnLib is using for finding counterexamples is the partial W-method or also known as Wp-Method, which follows the basic principles of the W-method. However, the main difference between these two methods is the size of the generated test set. In Wp-method a smaller set is constructed by applying a two-phase technique for test case generation: In the first phase we derive a test set from the *state cover set* and the *characterisation set W*. In the second phase we create another test case which is derived from the transition cover set P and the identification sets W_i. The characterisation set W as also the transition cover set P, have been already described in the W-Method. The *state cover set* as also the identification sets W_i will be described below.

Definition [52]: Given an FSM M with Q as the set of states, an identification set for state qiinQ is denoted by W_i *and has the following properties:*

1. The identification set is a subset of W

2. For each state other than qi, there is a string in W_i that distinguishes qi from qj
3. W_i is minimal, no subset of W_i satisfies the previous property.

A state cover set is a set of strings where, when starting from the initial state s_0 you can move to each other state and reach any other state. For example the string that you apply for moving from state s_0 to state s_1 belongs to this set as also the string that you apply for moving from state s_0 to state s_2, s_0 to state s_3, s_0 to state s_n. For simplicity reasons, we assume that m=n, where m is the number of states in M and n the number of states in the correct design, as defined before. In addition, we define that the test case T consists of two subsets T_1, T_2, where $T = T_1 \cup T_2$.

The following steps are performed using the Wp-method for computing the subsets T_1, T_2 [52]:

- Step1: computation of the transition cover set P, the state cover set C, the characterisation set W and the state identification sets W_i for M.
- Step2: $T_1 = C \cdot W$
- Step3: assume W is the set of all state identification sets of M, where $W = \{W_1, W_2, ...W_n\}$
- Step4: assume $R = \{r_i 1, r_{i2}, ..., r_{ik}\}$ is the set of all sequences that exist in the transition cover set P but not in the state cover C, so $R = P - C$. Moreover, $r_{ij} \in R$ such that $\delta(s_0, r_{ij}) = s_i j$
- Step5: $T_2 = R \otimes W = U_{j=1}^{k}(\{r_{ij}\} \cdot W_{i,j})$, where $W_{ij} \in W$ is the state identification set for the state s_{ij} and this $\otimes$ operator stands for the *partial string concatenation operator*

In the first phase, the mealy machine M^h (hypothesis) is tested against an element of the T_1 set (so is tested each state for equivalence with the corresponding M) [52]. In the second phase, the set T_2 checks the remaining transitions in M^h against M.

3.1.1 The MAT model interfaces in LearnLib

In Section 2.1.1, the MAT approach has been described. The main components of the MAT model are implemented as interfaces within LearnLib and

will be presented below [56].

The main component of the MAT model is the learning algorithm. Within LearnLib, the *LearningAlgorithm* interface provides the implementation of existing learning algorithms and can be used as basis for starting and applying the learning procedure. Functions and objects that these algorithms need, are provided within this interface. Functions used for queries generation as also methods used for connecting to the SUL and defining the accepted alphabet, are also parts of this interface.

Another MAT component, implemented in LearnLib, is the Automaton itself. LearnLib provides a library where already the Mealy Machine as also the DFA automaton are implemented. Therefore it can be used without more developing effort. The alphabet as also the learning algorithm that will be used should be defined from the user. The Alphabet is also deployed as interface within LearnLib and can be used.

Lastly, the two Oracles used at the MAT approach are also implemented and provided within LearnLib: the MembershipOracle for answering learning queries and the EquivalenceOracle for evaluating the hypothesis through approximation methods.

In Figure 3.1, some of the MAT model components, which are implemented in LearnLib as interfaces are presented. As illustrated, the MAT model is completely supported within the LearnLib infrastructure. The Learner side is supported in LearnLib by the "LearningAlgorithm" interface, which sends Membership Queries and Equivalence Queries to the Teacher side. The Teacher side in LearnLib implements the two Oracles (Membership and Equivalence), which answers the queries sent from the Learner.

The counterexample analysis in LearnLib is handled from the *CounterExampleHandler* interface. The Teacher and Learner components are responsible for answering and asking Equivalence and Membership queries, which provide information about the SUL, whereby the counterexample analysis follows different rules and does not depend on the SUL behavior. Therefore, the counterexample analysis is a separate component and is not included within the Learner or the Teacher components. So practically, two components, the Learner and the counterexample analysis component are queuing

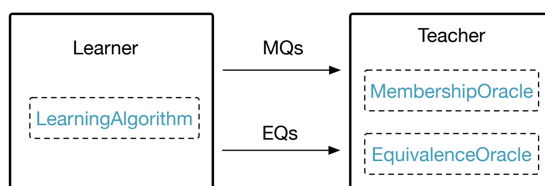

Figure 3.1: The MAT Model components as related to the LearnLib interfaces (dashed borders), source [56]

the Teacher with Membership queries, since the Equivalence queries are being approximated by new added Membership queries [56].

3.1.2 Mapper

In Figure 3.2 is illustrated how LearnLib is communicating with the SUL by sending input symbols to it. The SUL responds to these input symbols by producing output symbols which are sent back to LearnLib. Learning algorithms however, such as the L* that is used in this master thesis, can be effective with a limited number of input and output symbols [20]. However some systems with higher complexity, such as protocols, are using a lot of parameters and variables for connection establishment and synchronisation. For that reason, it is possible to reduce the number of parameters, by applying abstraction techniques. The implementation of a *mapper* that is placed between the Learner (in our occasion LearnLib) is responsible for this role. Through the abstraction, the number of the parameters passed to/from the Learner to/from the SUL is reduced.

For our work, we have used this abstraction method, which will be presented in Section 5 by creating a mapper component and placing it between the

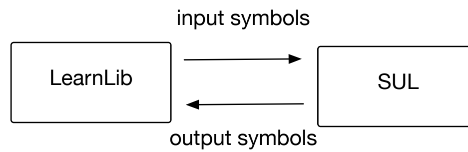

Figure 3.2: Basic Setup

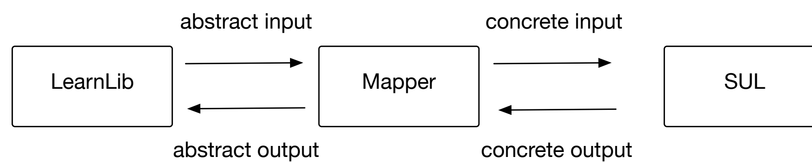

Figure 3.3: Mapper

Learner and the SUL. In Figure 3.3 is illustrated how the mapper is placed between the Learner, which in this case is LearnLib and the SUL. The mapper abstracts the concrete outputs, which are sent from the SUL and sends the abstracted outputs as inputs to the Learner. On the other side, the Learner sends abstract input symbols to the mapper, which maps them to concrete ones and sends them as input to the SUL. When a reset message is sent from the Learner to the SUL, a reset message is also sent the same moment to the mapper component. The basic element of the mapper is the abstraction function. As stated in [5], a mapper is also considered to be a Mealy machine, which has as inputs and outputs the concrete and abstract sets, with a set of mapper states, with an initial mapper state with a transition function and also an abstraction function.

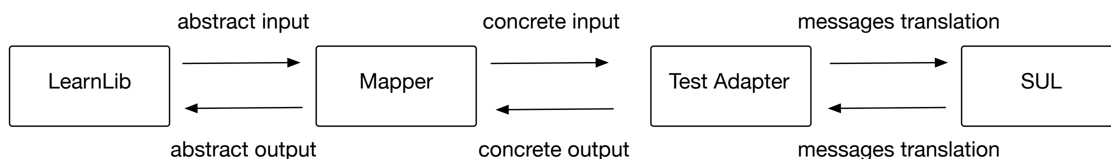

Figure 3.4: Test Adapter

3.1.3 Test Adapter

As described in the previous section, the mapper is responsible for the mapping between abstract and concrete values between LearnLib and the SUL. However, the concrete values that are produced by the mapper have to be passed to the SUL in a way that the SUL understands them. Therefore, they have to be transformed in the right format that can be read by the SUL. For that reason, another component that is called *test adapter* has to be placed between the mapper and the SUL. As shown in Figure 3.4 the test adapter receives the concrete values from the mapper and translates them to an understandable format for the SUL. The mapper and the test adapter components are not per default implemented from the LearnLib library; they are depending each time on the SUL's methods and therefore have been manually implemented. In Section 5 the practical aspects of the mapper and test adapter components, which have been created for the learning process, are in detail described.

3.2 LearnLib Studio

As aforementioned, LearnLib provides different learning algorithms. Their main difference is how they collect knowledge about the system under learn. On top of that, there are also differences on the number of membership and equivalence queries, on the size of these queries etc. For this reason, a configuration profiling - tool is needed in order to inspect and evaluate

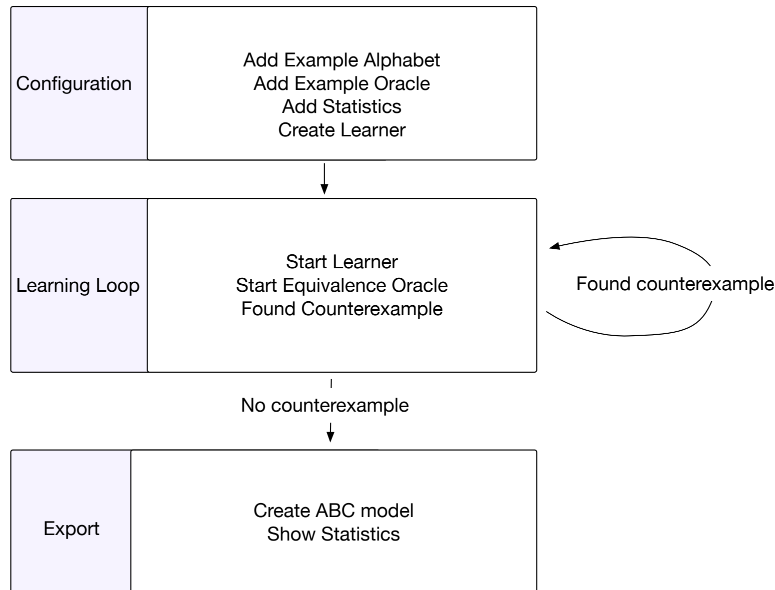

Figure 3.5: Model learning setup in LearnLib Studio. Source [56]

these differences. This profiling tool is called LearnLib Studio and shows how the different learning algorithms perform in practice by providing statistics about their behavior and efficiency [65].

LearnLib Studio is basically the graphical interface of LearnLib for designing and executing learning setups [47]. This learning setup can vary from learning algorithms, system adapters, to query filters, abstraction providers, etc. LearnLib Studio is based on jABC [74] technology and presents the LearnLib components available as modules in an executable graph [47].

In Figure 3.5 is shown how the learning setup has been initially modelled and used in early versions of LearnLib Studio [56]. In particular, all the components which take part on the learning procedure are configured and initiated in the “Configuration” phase. After that, the actual learning starts

and a hypothesis is constructed. In the last phase, the results of the "Learning" execution phase are exported and statistic results are shown. Nowadays, this approach has been simplified and improved, without requiring so many steps and configurations.

To sum up, LearnLib is a powerful and robust tool for fast active automata learning, which is implementing the MAT model. It has been used by several researchers and in many projects, as for example for inferring models of bank cards [4]. In [20], LearnLib contributed in learning communication protocols. Moreover, in the technical university of Dortmund models of a web application has been generated by using LearnLib [83]. Another tool called Tomte, which will be presented in the next Section 3.3, is using inference algorithms from the library of LearnLib in order to construct model abstractors [78].

3.3 Tomte

Tomte is another tool which can be used for active learning of automata. It is a tool that has been implemented from Fide Aarts, Paul Fiterau-Brostean, Harco Kuppens and Frits Vaandrager and analytically demonstrated in their paper [7]. Although we do not use Tomte in our experiments, it worths mentioning this tool since it is a powerful tool in the field of automata learning. Besides, it implements an algorithm for automatically constructing abstraction mechanisms.

The learning of automata can be performed from the Tomte tool, which automatically constructs abstractions of the system under test and send them to the Learner [78]. Tomte is placed between the Learner and the system under test and acts as a mapper, which receives an abstract input from the Learner and constructs a concrete input to be given to the SUL. In the same way, it constructs abstract outputs to be passed to the Learner, based on the concrete outputs received from the SUL. Tomte is using LearnLib for representing the Learner which follows a nondeterministic behavior. As SUL, can be used either a given real black-box system or a simulated model, which can be constructed with a so called "SUT tool" [77]. The

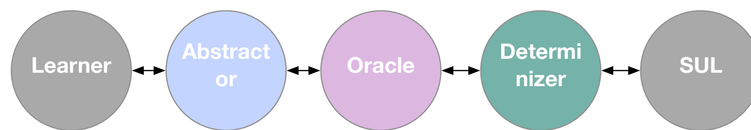

Figure 3.6: Architecture of Tomte

fully architecture and the components of which Tomte consists, are the ones visualised in Figure 3.6:

The Learner, in case of Tomte, is implemented by the learning tool LearnLib, which deploys several learning algorithms. Tomte is implemented in the following way: between the Learner and the SUL are placed three components which communicate in both directions with each other [7]. The first one is called Determinizer and is used for reducing the nondeterministic behavior that may be produced from the SUL. The second component is called "Lookahead Oracle" [7] which "remembers" the values of the already passed data, by storing all the traces of the SUL, which have been observed in the learning phase, in an observation tree. Lastly, the third components called "Abstractor" acts as a mapper, which translates the concrete outputs of the SUL to abstract inputs to be passed to the Learner. Therefore, any deterministic automaton can be learned from the Tomte tool. As presented in [78], the Tomte tool has successfully automatically learned real reactive systems such as the biometric passport and the SIP protocol.

4 Network Protocols

"Imitation is the sincerest flattery [2]. "

[Charles Caleb Colton, English writer, 1780–1832.]

The two network protocols that we focus on in this thesis are the Heartbeat Protocol and the File Transfer Protocol. Both of them have been used in the Internet communication and are following some base standards as described in RFC [66]. In this section these two protocols, which constitute the systems under learn (SUL), will be presented and in detail described.

4.1 File Transfer Protocol

The FTP is a server-client protocol that enables the transmission of file contents from a client to a remote server [23]. Through the FTP Protocol, the client can upload, delete, download, copy, rename and move files that are located on the server side. Therefore, the reliable sharing of files and data, as also the remote storage of them can be achieved through the FTP.

As defined in the official RFC description of the FTP protocol [23], FTP is an end-to-end protocol, which is used for reliable communication between two peers. It is making use of the transmission control protocol (TCP) for enabling the network communication [51]. TCP is a protocol standard which specifies how to establish and maintain a network connection for exchanging data between two hosts [72]. Thus, FTP is based on the TCP protocol when initiates a connection between a client and a server. This connection is initiated by the so called *control connection*, which should be established for exchanging commands and replies between the two peers. This connection follows the standards of the Telnet protocol and handles

(controls) the communication between the server and the client. After the initialisation of the control connection, the *data connection* can be established, which is a duplex connection for transferring data. These two connections form the FTP base functionality.

The FTP service runs per default at the port 21 but this port can also be manually configured and changed by the user. FTP commands are sent from the client to the server and FTP replies are sent from the server back to the client. Initially, the control connection is initiated by the client sending the connection parameters to the server. More specifically, these connection parameters consist of the internet address of the server and of the port that the FTP service is running. After the control connection has been initiated, the authentication phase occurs, where the client sends its credentials in order to be authenticated to the server. After the "connect and authenticate" phase, a data connection can start. FTP commands define the parameters of the data connection, which can be each time configured [23].

In Figure 4.1 the FTP architecture is presented based on the RFC official FTP specification. The DTP Server/User stands for Data Transfer Process and is responsible for creating the data connection. During the transfer of files, the DTP is also responsible for managing the data connection. The PI Server/Client stands for Protocol Interpreter and its main role is the transfer of the commands received from the control connection to the DTP. Additionally, on the client side a user interface is deployed (GUI), which communicates with the protocol interpreter.

As briefly mentioned above, the basic connections that the FTP establishes are the following:

- The Control Connection

  The control connection, which follows the Telnet protocol, acts as a communication path between the client (user-PI) and the server (server-PI). Within this connection, commands and replies are exchanged between the client and the server. Let's assume the following simple scenario, where the client wants to communicate to the server. Initially, the client opens a TCP connection from a random port to the server's command port (which is usually the port 21). The client can

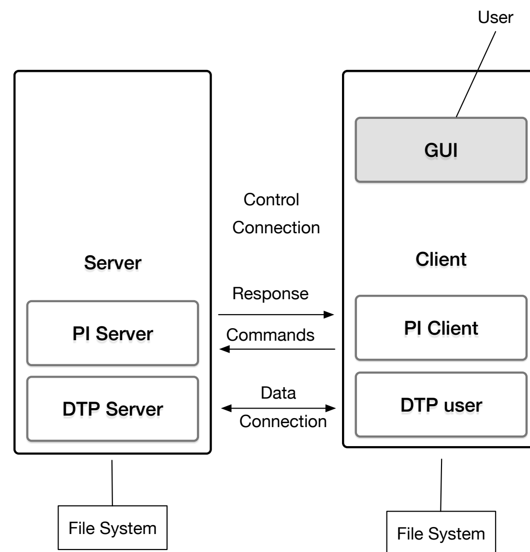

Figure 4.1: FTP architecture

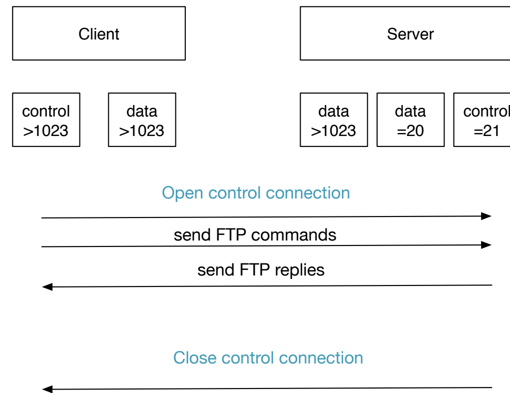

Figure 4.2: FTP Control Connection

then send commands to the server and the server can reply back by generating command replies. At any point the server can close the control connection, after the client has requested a "connection close" and after all the transfers and replies have been completed as described at the protocol's specifications [23]. Furthermore, after the control connection has been initiated, the transfer of data can start. During this exchange and transferring of information the control connection should remain open. That results to the fact that without a control connection, no data connection can be initiated [24]. In Figure 4.2 the control connection is visualised.

- The Data Connection

  The data connection is initiated between the client (user-DTP) and the server (server-DTP), after a control connection has been established. It acts as the communication channel for exchanging real data. The data connection indicates that data can now be exchanged between the server and the client. There are two different modes, the FTP protocol

can be configured: the active mode and the passive mode[23]. On the active mode the server starts and closes the data connection, whereby on passive mode these actions are performed by the client. The default mode of the FTP protocol is the active mode. If the client would like to use the passive mode, a "PASV" command should be sent to the server.
In the active mode the client (user -PI) sends to the server (server-PI) the port where the server should connect to for setting up the data connection. The client sends this information by connecting first to the default command port (port 21) where the control connection has been initiated. After that, the server establishes the connection from its local port (port 20) to this client port that has been sent before from the client and the client already is listening to and waiting for the connection [24].
Since it is not so practical that the server has to initiate the connection, the passive mode is more often use. By sending the client a "PASV" command to the server, it indicated that the client will establish the connection. The Server reacts to this command, by opening a random port and start listening to it. The client connects and initiates the data connection at this server's port [24].
As stated in RFC [23],*"all FTP implementations must support the use of the default data connection ports, and only the user-PI may initiate the use of non-default ports"*.

Transfer of Data

After the data connection has been initiated, the transfer of the data can start. The files can be transferred only when the data connection has been established. FTP uses the *MODE* command for defining the way that the data will be transmitted and the *TYPE* command for defining the type (representation) of the data that will be transferred. The default type in FTP is the *ASCII TYPE* [23]. However, FTP does not support all the kind of data type representations, therefore the two peers should first perform the corresponding changes in the data representations before starting sharing them. In addition to the different data types provided by FTP, the structure

of a file can be also defined from the client-user. The three file structures that the FTP supports are [23]:

- file-structure: where the file is just a sequence of data bytes
- record-structure: the file consists of sequential records
- page-structure: the file consists of independent indexed pages

The default FTP structure type is the file-structure.

Additionally, the FTP distinguished between different type of commands; access control commands, transfer parameter commands and FTP service commands [23]. For this master thesis purposes, we are focusing on the access commands, which are used in the authentication and the login phase. In particular, the *access control commands* that we are focusing on are the following:

1. User name (USER), for user's authentication
2. Password (PASS), for user's authentication
3. Change working directory (CWD), for changing the current directory
4. Logout (QUIT), for requesting the closing of the control connection

   Additionally to these, we use from the *transfer parameter commands*, the
5. Data port (PORT) command, for specifying the internet host address and the TCP port address of the FTP server to be connected and form the *FTP service commands*, the

6. Store (STOR) command, for uploading a file to the server side.

After the client has sent on of these three types of commands to the server, the server responses with an FTP reply. The FTP replies are represented as a string which contain three digits and some text. The text is only relevant for the user, by giving him/her some meaningful information. The digits are relevant for the user-PI and each number combination has also another meaning. In particular, the first digit denotes if the FTP reply is good, bad or incomplete [23]. The second digit gives information of what kind of error has occurred, for instance file system error. The third digit gives a better explanation of the error described from the second digit.

We illustrate the following example: The first digit "1xx" indicates a "Positive Preliminary reply", the second digit "x2x" refers to replies that are related

to the control and data connections and the third digit “xx5” gives more information about the control or data connection which was specified in the second digit. As a result, the three digit number “125” would mean the following server reply: “Data connection already open; transfer starting” Different FTP implementation should strictly follow the digit combinations (codes) that are defined at the RFC standard [23] and not invent their own. The text that follows after the three digits is not a compulsory field but it is recommended from the FTP standard, since it gives meaningful information to the user.

Based on the protocol’s requirements, every client command should generate at least one server reply, so that the other peer always knows the state of the server [23]. This reply is called primary server reply and is a mandatory action. However, the server can also generate a secondary reply if the client commands require a second reply. For example, a second reply could be in case of a file transfer, a reply for reporting the progress of a file or for closing the data connection.

4.2 Heartbeat Protocol

In this section, an extension of the Transport Layer Security protocol, called Heartbeat protocol will be presented. Before providing information of this protocol, the understanding of how the TLS protocol works will be described, since it is required as an a-priori knowledge at this thesis.

The TLS Protocol

Most of the services over Internet are making use of the so called "OpenSSL" protocol in order to transfer data over the Internet securely. In particular, OpenSSL is a popular open source project for secure transmission of data over the Internet. It provides a cryptographic library which implements the SSL and TLS protocols [62]. SSL stands for Secure Sockets Layer, where TLS stands for Transport Layer Security. Both protocols provide data encryption and authentication between the two peers communicating. SSL protocol is the pioneer in secure communications protocols and the basis for the TLS protocol. However, the SSL protocol has never been published by the Internet Engineering Task Force (IETF) and no formal specification of it has been described [76]. Therefore, TLS has been introduced as a standardised protocol, which will allow client/servers applications to communicate securely.

TLS Handshake

A connection between a client and a server is always initiated with a handshake action between them. The purpose of the handshake is to provide both sides with the secure elements that will be used within this communication session [80]. The handshake phase is summarised as follows, as described at the official TLS documentation [80] :

The TLS client sends a "client hello" message to the TLS Server and also some cryptographic information. The server responses with a "server hello" message and sends also his certificate and his cryptographic information

to the client side. The client verifies the server's certificate and the cryptographic information that were sent. After that, in case the verification succeed, the client encrypts with the server's public key the common key that will be used at this session and sends it to the server. The server decrypts the session key with its private key. Both send a "client/server finished" message to indicate that the handshake phase is over and they can start exchange messages, encrypted with the shared session key. In that way confidentiality of data has been achieved.

Heartbeat Protocol, an extension of the TLS protocol

The Datagram Transport Layer Security (DTLS) is a specific category of the TLS protocol, which provides communication security for datagram protocols. According to RFC [79], it is designed for securing the communication traffic between two peers which occurs when unreliable transport protocols are used. The problem with such protocols is the fact that they have no session management. When the DTLS needs to know if a peer is still alive, it has to perform a renegotiation between the two peers. For avoiding this costly procedure, an extension, called Heartbeat, of the TLS and the DTLS protocol has been introduced, which is used to check whether the connection between two peers (server/client) is still "alive". Precondition to this is that the two peers communicate with each other over a TLS connection, since the Heartbeat protocol runs on top of the TLS Record Layer. The client and/or the server can send a heartbeat message for checking whether the other peer is still "alive".

At the point of the initial TLS handshake, the peers should demonstrate if the Heartbeat extension is supported and will be used. The support of the Heartbeat extension is indicated with "Hello extensions". For example, a peer can have the choice either to receive HeartbeatRequest messages and respond with HeartbeatRequest messages (peer_allowed to_send as the HeartbeatMode) or only to send HeartbeatRequest messages (peer_not_allowed_to_send as the HeartbeatMode) [79]. In case a peer is indicated with peer_not_allowed_to_send, HeartbeatRequest messages must not be sent to this peer. On the latter case, both peers can send a HeartbeatRequest message in order to verify if the connection between them

is still present and a HeartbeatResponse message as a reply to the request to indicate that the peer is still alive. These messages consist of the following fields, according to the RFC description [79]:

- Type, 1 byte : the type of the message, either HeartbeatRequest or HeartbeatResponse
- Payload Length, 2 bytes : the length of the payload
- Payload, 65535 bytes : the payload which consists of arbitrary content
- Random padding, at least 16 bytes : the padding of a received Heartbeat message must be ignored

If a HeartbeatRequest message has been sent but no HeartbeatResponse has been received then the HeartbeatRequest message is still *in flight* as it is called [79]. At that point, the other peer should not start sending other messages when a HeartbeatRequest is in flight, since they will be discarded. It is also important that a HeartbeatRequest message should not be sent during handshakes. As described in RFC, the following restriction should be taken under consideration:

- Payload of HeartbeatRequest must equal the payload of Heartbeat Response
- Payload length of HeartbeatRequest must equal the actual payload length
- When a HeartbeatRequest is sent, a Heartbeat Response must be received
- The payload length must not be greater than 2^{14} bytes
- Another HeartbeatRequest message should not be sent while there is already a HeartbeatRequest message in flight
- Only one HeartbeatRequest is allowed to be in flight at a time

What is actually performed by the protocol is illustrated in Figure 4.3. When a request has been received from one peer, the other end-point responds to that with a message that echoes back the HeartbeatRequest payload and its own random padding. In that way, is indicated that the peer is still alive.

The Heartbleed Vulnerability

The OpenSSL implementation of the Heartbeat extension as described above, contained a vulnerability where an attacker could retrieve valuable information like history session, keys, passwords etc. from the user [18]. The attack is described as follows: With the Heartbeat extension of the OpenSSL protocol, it is possible as described above to get informed if the server is still alive. Therefore, the client sends for instance the HeartbeatRequest "example" as shown in Figure 4.3 to the server and claims that the length of this request is 20 bytes. However, the actual length of this message is 7 bytes. The problem lies to the fact that the length of the HeartbeatRequest is not being checked and verified from the other peer. Thus, the server replies with an 20-byte HeartbeatResponse by mirroring the HeartbeatRequest "example". In order to complete the remaining 13 Bytes, the server pads 13 Bytes of its heap memory which can contain private data. These data can be username, passwords, encryption keys, etc. In addition, if the attacker could retrieve a history session from the server, then these leaked keys could be used to decrypt all the information from the history session.

In a closer more technical view, the maximum HeartbeatRequest size is 16KByte including one byte for identification of the message type and two bytes for defining the length of the Heartbeat request message. As a result, the payload and the padding together should not be greater than: $2^{(14)} - 3 = 16381$ bytes. The Heartbeat response sends an exact copy of the received request message back to the sender. First it is checked from the receiver if the responded message is of type HeartbeatRequest and then extracts the payload length of it. After that, the message is allocated to the memory for the HeartbeatResponse. It copies the payload from the request message to the response and sends it back to the sender. This is illustrated through the following line:

```
memcpy(bp,pl,payload)
```

Hence, this piece of code indicates that if the payload length is bigger than the actual message then not only the payload from the heartbeat message will be copied to the HeartbeatResponse but also what lays after that in the

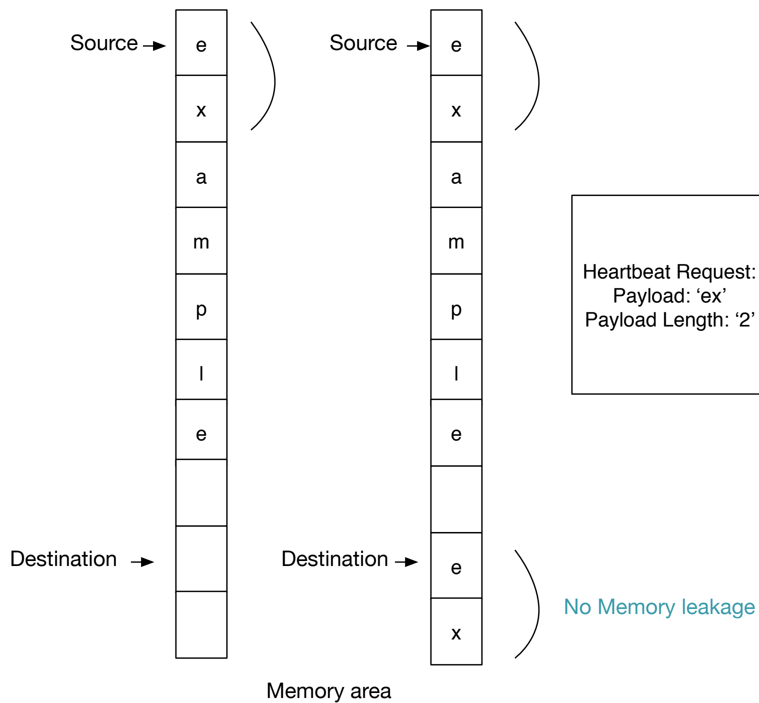

Figure 4.3: Heartbeat Messages

memory. In Figure 4.3, an example of no data leakage is presented, whereby in Figure 4.4 a data leakage from the memory is illustrated. The maximum data leakage could be up to 65535 bytes, which is the maximum payload length that is allowed by the OpenSSL implementation.

The fix of this bug could be summarised within just two additionally lines of code, where this check is performed:

```
if (1 + 2 + payload + 16 > s->s3->rrec.length)
return 0;
```

Although the fix of the bug is straight forward, the impact of this vulnerability is severe.

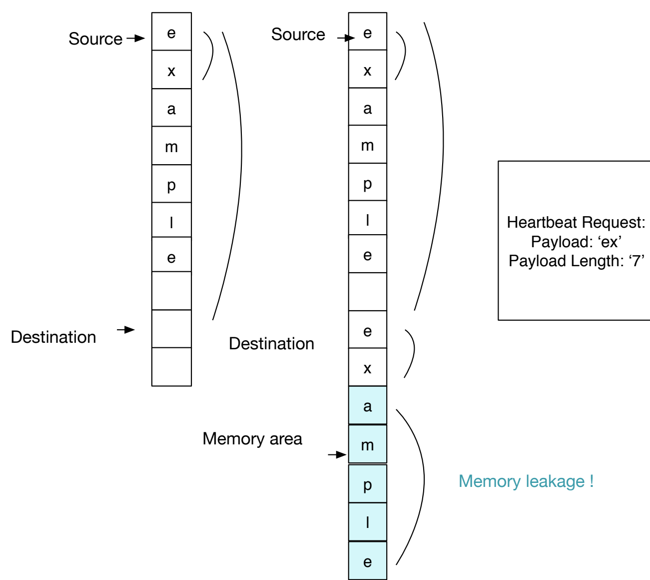

Figure 4.4: Memory leakage

5 Active Learning of Network Protocols

"A theory can be proved by experiment; but no path leads from experiment to the birth of the theory [85]. "

[Albert Einstein]

Learning Setup for all the SULs

For our learning setup, we have use the LearnLib tool, a framework for learning systems, which are represented as an automaton. LearnLib as described in Chapter 3.1, already includes a variety of learning algorithms, for performing active automata learning technique to the SULs. For our purposes, we have chosen the most well known learning algorithm, the L* Algorithm from Angluin [9], as the main learning algorithm. In addition, we have used two different oracles (one at each learning phase), which is used for answering equivalence queries. The idea is to compare the performance of the two different oracle implementations and find out which one performs better. The oracles that we have used are: the 'RandomWalk Equivalence Oracle' and the 'W-method', that have been described in Chapter 3.1.

In particular, the SUL is an implementation of the protocol that is analysed. The Learner (in our case LearnLib) and the SUL are connected to each other and exchange the following information: The Learner sends queries to the SUL, by giving some input symbols to it and the SUL responses to these queries, by producing some output symbols and send them back to the Learner. After the Learner has finished sending the first set of inputs, a reset message is sent to the SUL and the SUL is reseted (set to the initial state). The set of inputs that we give to each SUL is analytically described in

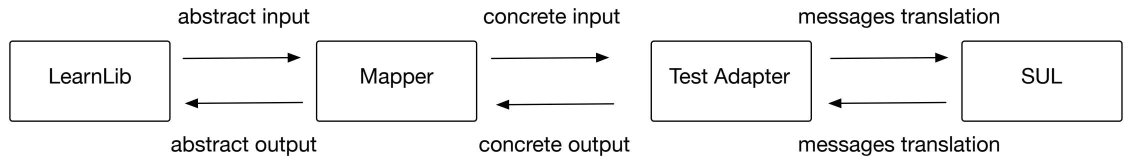

Figure 5.1: Our basic learning setup

Section 5.2.2 for the FTP protocol and in Section 5.1 for the Heartbeat protocol. The Learner observes each time the outputs of the SUL. Based on these observations, a hypothesis H is constructed. The hypothesis is evaluated by performing the *W-method* and the *Random Equivalence* method, for building equivalence queries and querying the SUL. In case the hypothesis is correct, the learning has finished, otherwise a counterexample will be produced and the hypothesis has to be refined. In addition to that, for this communication between the Learner and the SUL, a mapper and a test adapter has been placed in between. The functionality of a mapper and a test adapter has been described respectively in Sections 3.1.2 and 3.1.3.

We reset the corresponding SUL after every trace sequence of inputs has been sent to the SUL and we set the transition function to its initial state (state 0), before we start sending the next input sequence. That is to say that when a new sequence of inputs is received, the SUL as also the mapper should be in their initial state. We implemented the reset function by simply creating a new instance of the SUL every time a reset is triggered. Besides, we use the same learning setup for learning all the systems under learn. A graphical model representation of our basic learning setup is illustrated in Figure 5.1.

Figure 5.1 shows how we have connected all these components to communicate with each other. The LearnLib component is actually the Learning algorithm which is already implemented as interface in LearnLib. We implemented the Mapper component and the test adapter component as also the SUL in case of the Heartbeat protocol. Thus, in the next section we will describe how we have implemented the aforementioned components

for our learning setup. We present the test adapter in the next section as it follows the same logic for all the learning setups. The Mapper will be described separately for each protocol and for each client implementation, because it is important to explain how we abstracted and then concretised the inputs/outputs given/produced to/from each SUL. Lastly, for the FTP protocol specifically we describe the client(Section 5.2.2) and the server setup (Section 5.2.1) since we have connected a real system to LearnLib, which is not the case for the Heartbeat protocol because we have set up our own trivial implementation of it. Lastly, issues that have been introduced with our initial setup will be discussed at the last section as also tunings and small adjustments that have been done in order to overcome these difficulties.

Test Adapter in the Learning Setup

In order to connect the Learning algorithm directly to a real system (in our case protocols) we need a so called test adapter. The reason for that is that the learning algorithms in general are using alphabets [45]. A learning algorithm expects to receive words as input that belongs to an alphabet that we initially define and will produce also words as an output alphabet. Thus, in order to connect the L* Algorithm to our real system, we have implemented a test-driver, which translates the word input into sequences of real actions (for instance method calls) and vice versa; the real outputs (fore instance return values from the methods) are translated into output words. In the implementation of our test-driver we have implemented a method which translates input words into method calls and the return values of method calls into output words.

What we have implemented, is the translation of the concrete messages that are produced by the mapper in a format that the SUL understands them. For instance, for the Heartbeat protocol, the test adapter receives the concrete messages from the mapper (for example a HeartbeatRequest packet) and calls the *SendHeartbeatResponce* method from the SUL with parameter the HeartbeatRequest packet. The Heartbeat protocol responds with a return value, which the test adapter processes and translates to a concrete output sequence (for example a HeartbeatResponce packet),

which passes over to the mapper. For the FTP protocol the same logic is applied. For example, the concrete input message *Connect_ hostname_ username_ password* will be translated from the test adapter to the following function call: $ftp.connect(host, username, password, 21)$, assuming that the default port has been set to 21.

5.1 Heartbeat Protocol

The first system under learn (SUL) that we analysed in this thesis is the OpenSSL Heartbeat Protocol and the main focus targets on revealing the “Heartbleed Vulnerability” [73] of it by using learning techniques. In particular, by learning the behavior of the correct implementation and the behavior of the faulty one and by comparing these two learned models, we should be able to observe the unexpected behavior of the protocol and uncover this serious vulnerability. As described in Section 4, OpenSSL is an open source library that provides an implementation of cryptographic protocols such as the Secure Socket Layer (SSL) and the Transport Layer Security (TLS) protocol. We attempt to model the behavior of the Heartbeat protocol, which is considered a black-box implementation and can be inferred as a Mealy machine.

The basic experimental setup that we have used, is based on the learning principle: The Learner (LearnLib) sends inputs to the SUL and the SUL responses to these inputs. Therefore, the learning algorithm sends HeartbeatRequests to the Heartbeat implementation and it responses with a HeartbeatResponse. In-between is placed the mapper, which abstracts the sent and received messages and the test adapter which translates them to a format, which is understandable from the SUL. However, we did not plugin and trigger the ’real’ Heartbeat implementation itself but we implemented the behavior of the two implementations to be learned: the one with the vulnerability and the correct one. In that way we were able to simulate the protocol’s behavior and set it under learn. The reason why we set up our own implementation of the protocol is based on two facts: (a) we wanted to save time and effort for our learning set up (it is more complicated to set up a client and a server that communicate via SSL and then perform the

learning only in the Heartbeat part of this SSL communication, than just simulating the Heartbeat protocol) (b) we wanted first to investigate if the Heartbleed vulnerability could be revealed with active automata learning. In case we were able to prove it, then we would connect the real Heartbeat implementation to our Learner.

Primarily, the focus has been given on the abstraction of the HeartbeatRequests and HeartbeatResponses, based on validity check operations, as will be described in the following subsection. By constructing two learned models; both for the correct and the faulty implementation, derived by applying learning algorithms and afterwards performing a comparison between the two models generated, we tried to testify the compliance of the Heartbleed protocol to the model specification.

5.1.1 Mapper for Heartbeat Protocol

As mentioned above the messages sent and received (HeartbeatResponse, HeartbeatRequest) from and to the SUL, have been evaluated to valid or invalid ones. For this operation, a mapper component has been used as described in Chapter 3.1.2. The main functionality of the mapper component is the mapping of the parameters which are included in the exchanged messages from a concrete to an abstract domain and vice versa [20]. We have abstracted the functionality of the Heartbeat protocol by setting boolean flags for abstracting some parameters, as will be described in this section and also by performing validity checks in order to distinguish between a valid/invalid request and a valid/invalid response.

From abstract Input to concrete Input

In the Table 5.1 the abstraction of the input messages through the mapper component is visualised. The abstracted inputs that are sent from the Learner to the SUL, are concretised based on the operations as shown at this Table. The Learner sends a HeartbeatRequest message and the SUL responds with a HeartbeatResponse. Actually in our case the Learner is like the client that sends requests and the SUL is like the server that answers with responses.

HeartBeat Protocol	
Abstract Input	**Concrete Input**
Valid HBRequest	Message as sequence of bytes where: FirstByte=1 & GivenLength = HBRequestLength
Invalid HBRequest	Message as sequence of bytes where: FirstByte != 1 || GivenLength != HBRequestLength
peer_allowed_to_send(true)	1
peer_allowed_to_send(false)	0
message_in_flight(true)	1
message_in_flight(false)	0
IsHBRequest	FirstByte=1
IsHBResponse	FirstByte=0

Table 5.1: mapper for Heartbeat Protocol (Inputs)

HeartBeat Protocol	
Abstract Output	**Concrete Output**
Valid HBResponse	Message as sequence of bytes where: FirstByte=0 & HBResponse = HBRequest & HBResponseLength= HFRequestLength
Invalid HBResponse	Message as sequence of bytes where: FirstByte!= 0 || HBResponse != HBRequest || HBResponseLength != HFRequestLength

Table 5.2: mapper for Heartbeat Protocol (Outputs)

Initially, five parameters are sent from the Learner to the SUL:

- the message (HeartbeatRequest)
- the content type of the message (if it is a HeartbeatRequest or a HeartbeatResponse)
- the length of it (we call it *given length*)
- two more flags, which can have the value "1" when true or "0", when false. The flags indicate if the peer is allowed to send a message or not and also if a message is inflight or not.

Based on the RFC specification [1] a Heartbeat request message is in flight until a corresponding Heartbeat response message is received or until the time of retransmission expires. The content type of the message indicates if the type of the message, i.e a HeartbeatRequest or a HeartbeatResponse. In case the Learner sends to the SUL a HeartbeatResponse message, then the message should be discarded, since the Learner will send only HeartbeatRequests to the SUL. The opposite of course scenario applies from the SUL to the Learner, where only Heartbeat responses should be allowed.

In particular, a **Valid** HeartbeatRequest message is mapped to a concrete sequence of bytes, where the first byte will be 1, showing that this message is a HeartbeatRequest. Moreover, the total byte message length should equal to the other input "*given length*", in order to be a valid message. If the Learner sends as input to the SUL an **Invalid** HeartbeatRequest, that means that at least one of these two validity checks result to false. The mapper concretises these abstract inputs (Valid HBRequest, Invalid HBRequest, peer_allowed_to_send(true), etc.) to concrete values (Byte sequences, boolean variables, etc.) as shown in the Table 5.1 and sends them to the test adaptor which will call the corresponding method of the SUL with these parameters.

From concrete Output to abstract Output

Table 5.2, describes how the mapper translates the concrete outputs produced by the SUL to abstract outputs. On the other way around as before, the SUL now answers the previous query from the Learner, by producing a concrete output which the mapper abstracts either to a **Valid** or to a **Invalid**

HeartbeatResponse. In order that the mapper can decide if the concrete message from the SUL is valid or invalid, it analyses the validity of the message. We implemented our mapper, based on the Heartbeat specification and we describe how we performed the abstraction of the output in Table 5.2. Thus, we have defined the validity of the concrete output as follows:

If the SUL produces a concrete output of bytes, where the first byte is a zero (so type of message is HeartbeatResponse) and this sequence of bytes equals the HeartbeatRequest message sent before by the Learner and the length of the concrete output equals the HeartbeatRequest length that has been given before as input from the Learner to the SUL, then the mapper abstracts this concrete output to a valid HeartbeatResponse. In case the first byte of the concrete output is not zero (so the message is not HeartbeatResponse), or the HeartbeatRequest does not equal the HeartbeatResponse, or the length of this concrete output message is different to the previously HeartbeatRequest's length, then the mapper abstracts this output to an invalid message.

At this point we need to make a categorisation between the correct and the faulty implementation of the SUL. If we "query" the correct implementation, the HeartbeatResponse should be different as in the faulty one. It is important to mention, that we have connected the same mapper component to both SULs, so the learning set up is the same. The best way to illustrate the different reaction (output) of the correct SUL and the faulty SUL, for the same input, is through an example:

For instance the Learner generates an invalid request Message (Hello, 7) and sends it to the Mapper, where:

- Message: Hello
- Given length: 7 digits
- Actual length: 5 digits

The Mapper translates it to a sequence of bytes and sends it to the SUL, which is implemented without any security vulnerability (correct implementation). The SUL will testify if the given length equals the actual length and will conclude that they do not match. In that case, the implementation itself will "correct" the parsed parameter of the given length, by replacing it with

	Correct Implementation Heartbeat Protocol	**Faulty Implementation Heartbleed Protocol**
Valid Request	False	False
Request Equals Response	True	False
Valid Response	False	**True**

Table 5.3: Boolean Table for Heartbeat Response in Heartbeat Protocol

the correct parameter of the actual length. Thus, we define the following checks, which in that case will all result to false:

1. Valid Request: a request is valid when the length of the request message (five) equals the given length (seven) (illustrated in Table 5.1 with the check GivenLength = HFRequestLength)
2. Valid Response: a response is valid when the length of the response equals the length of the request (illustrated in Table 5.2 with the check HBResponseLength= HFRequestLength)

Additionally, when a HeartbeatResponse is sent, another check can be performed for checking the validity of the message:

- Request equals Response: a request message equals the response message, when the values are identical (illustrated in Table 5.1 with the check HBResponse = HBRequest)

On the contrary, if the same invalid request Message (Hello, 7) is sent to the faulty implementation, which does not comply to the Heartbeat specification, then the parameter "Valid Response" will result to True. This is due to the fact that the protocol will not perform any check of the parameter "given length" and hence it will generate a response message with the same length as the request message. Both scenarios are illustrated in Figure 5.3. Based on these verifications, the mapper can abstract the received HeartbeatResponses to valid or invalid ones and sends them back to the Learner.

5.2 FTP Protocol

The second system which is being exposed under learning is the FTP Protocol and in particular three different client implementations of it. Generally, the FTP server, which runs a server software, listens on a port for requests from clients that want to connect to the server. The client, which has an FTP client software installed, requests and initiates a connection to the server. When the client and the server are connected, the client can perform a lot of actions to the server, such as upload, download, delete files from the server, etc. The difference to the set up of the Heartbeat protocol is that now in our learning setup the FTP Server component is also added.

The different FTP Clients, which we have chosen as SULs, have been chosen in that way that they could give a variety of the system's behavior. SimpleFTP [3] is a very simple FTP Client, used for performing simple actions with the FTP Server. On the contrary, the FTPClient provided by Apache [26], is the most well-known and used FTPClient that incorporates the whole functionality of the FTP protocol. It performs all possible actions that could be accomplished from a client to an FTP Server. Last but not least, the FTP4jClient [25] is a library that also implements in Java the full-features of an FTP Client and supports different security authentications. It is interesting to examine how these various clients behave in respect to the specification of the FTP protocol as described at RFC [19].

In Figure 5.2, we present how the learning of the FTP protocol takes place. We give a short example in order to understand better how the different components are connected to each other and what happens in-between. Based on our diagram, we start in position 1, which is the Learner. The Learner sends the abstract input "Invalid Connect", which indicates that he wants to trigger an invalid connect. The Mapper translates this input to a concrete input which is: an invalid IP address (152.10.982.0), which does not exist, and the number 21, which is the default FTP port. The test adapter will translate it to an input that says to the SUL to call the connect function with the aforementioned parameters. The SUL will call the connect function which tries to connect to the server with an IP address that does not exists. Therefore, the server will never respond, since it is unreachable and the client implementation will throw a socket exception for

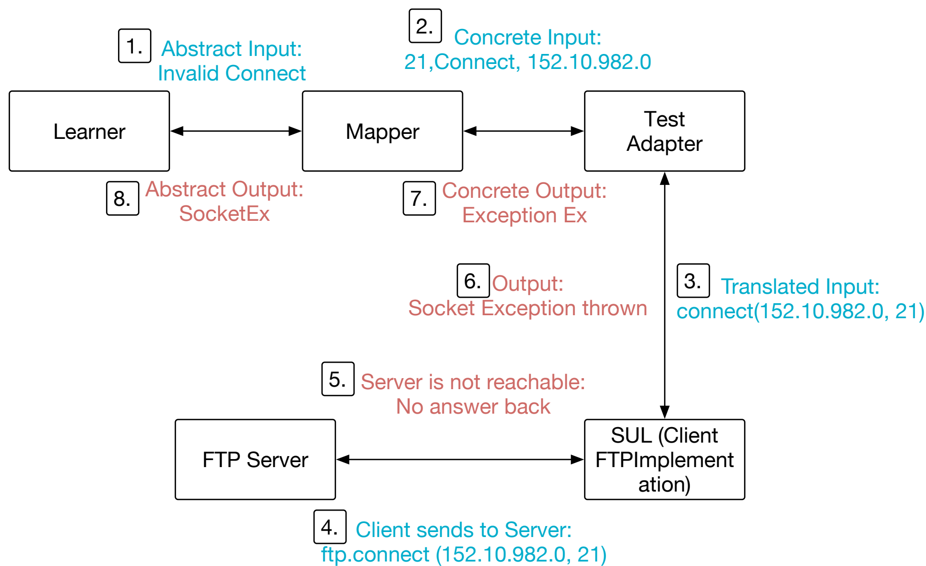

Figure 5.2: Example of our learning setup for FTP Protocol for the input *Invalid Connect*

example. This is how Learner, server and client communicate. The Learner triggers different functions of the client in a different order at each run, the client communicates with the server and produces an output based on this communication. The output is returned to the Learner, which tries to learn how the client behaves at each given input. Of course the behavior of the client is affected by the responses of the server as well.

In the next sections, the Server setup, which is common for all the FTP clients, as also the Client setup will be presented. The Client set up is different in each FTP Client implementation, therefore it will be demonstrated separately, within each FTP Client section.

5.2.1 Server Setup

The experiments have been performed in an Ubuntu 64-bit 14.04 machine, which runs as a virtual machine in an X Yosemite operating system version 10.10.5.

Regarding the learning of the FTP protocol, the FTP server that has been used for connecting to the FTP clients, is called 'very secure FTP daemon' or 'vsftpd'. More specifically, the version vsftpd-3.0.3 has been used for our experiments, which is the latest released version and is stated to be stable, secure and fast [82]. This server is an FTP server for Unix-like systems like Linux [82], which can handle virtual users and virtual IP configurations. According to that, we created three different virtual users with three different user-rights. These virtual users will be able to connect and login to the FTP server, move along their directories, upload or download files. The aim of this action, is to test how the FTP protocol handles situations where for example a user with no read or no write rights tries to access a folder. The users that have been created with different rights, in order to cover more cases, where such unauthorised actions take place. The users that we have created to be authenticated to the server are the following:

- virtual user1: This user has rights to upload and download files, under the users home directory: /var/www/user1. The home folder of the user is not writable only readable.
- virtual user2: This user has rights to upload and download files, under the users home directory: /var/www/user2. The home folder of the user is not writable only readable.
- virtual user3: This user has rights to upload and download files, under the users home directory: /var/www/user3. However, this directory is not set to "read only".

In the Table 5.4, the file permissions for each user are listed. The permissions are broken into 2 sections.

- the first section is denoted with the letter "d" for indicating a directory, with the letter "l" for indicating a link and with the symbol "-" for indicating a file.

Directory	user1	user2	user3
/var/www/userX	dr-x	dr-x	drwx
/var/www/userX/folder	drwx	drwx	drwx

Table 5.4: User's rights

- the second section consists of the next three spaces which indicate the read with "r", write with "w" and execute with "x" permissions for the owner of the file. For instance, the "r-x" sequence indicates that the user has only read and execute permissions, but no write permissions.

The Table 5.4 shows that the user1 and user2 have only read permissions at the root folder userX, where user3 has write and read privileges for his folder. However, for the subfolders of the root folders, all the users have the same rights; write and read. That is to say, that when for instance user1 logs in with his credential at the FTP server, he will be located under this path: /var/www/user1. This path should not be writable for user1, but only readable. Namely, the upload of a file should not be achievable. If the user1 changes to a subdirectory, the upload of a file should be realisable.

In addition, the settings for the vsftp server has been modified as follows:

1. Local users are allowed to log into the system.
2. The creation of virtual users is permitted.
3. Local users are restricted to their home directories

We have used the default configuration of the vsftp server but we have enabled additionally the above settings, in order to be able to create virtual users and also to restrict the rights of each user to his/her local area/folders.

5.2.2 Client Setup

After having described the Server setup for our experiments, we present the Client setup for each Client implementation in this section. The main component of the Client setup is the mapper and therefore we will describe it in detail for each FTP Client.

In the experiments that have been performed, five basic operations of the FTP protocol were used: connect, login, change directory, upload and disconnect. To be more precise, in order to achieve comprehensive results, all the systems under learn should have the same starting setup. In other words, we used the basic setup as described in Figure 5.2 and also we have given the same inputs to the SULs, in order to be able at the end of the experiments to compare the resulting models with each other. This can be achieved only when triggering the same methods every time in the three client implementations under learn.

Thus, the operations that we gave as inputs (with their parameters accordingly) to the SULs are the following:

- Connect: Connects the client to the remote FTP host.
- Login: Authenticates the user against the server.
- Change Directory: Changes to a new working directory on the FTP server.
- Upload: Uploads a file to the remote server.
- Disconnect: Disconnects from the remote server, optionally performing the QUIT procedure.

The last three operations are common to the three client implementations that we examine. The first operation differentiates only at the SimpleFTP to the other two clients, since it was not possible to perform two different calls for Login and Connect. The implementation of this client does not support two separate methods. Thus, only at SimpleFTP Clint with one function call, the user should give information for the server address to be connected and his credentials. Only one response which corresponds to both connect and login will be returned from the server.

Our mapper implementation is not the same for the three client implementations. Regarding the input sets, the mapper is almost the same in all three implementations, except one line of code difference for the SimpleFTP client, where connect and login is one operation and not two separate methods. This is expected, since we give always the same inputs to our SULs. Regarding the output sets produced by the SULs, the mapper defers in all the three implementations, because the concrete outputs produced from the SULs are completely different among the three implementations. Therefore, the

mapper acts differently for each implementation - various concrete outputs are mapped to various abstract outputs. Our test adapter implementation is not the same for each client, since it has to be adapted to the signature of the methods of each client; each client may have different amount of parameters for its methods and may produce different outputs (return values) for each method, that have to be translated from the test adapter.

In the next sections, we present the mapper for each client implementation. The test adapter is not described in a separate section since it does not hide any important functionality behind it; it puts the given parameters in the right order and calls the corresponding method from the SUL. For instance, the mapper passes to the test adapter the concrete input *Upload, filename* and the test adapter creates the input *ftp.upload(filename)* which will trigger the upload method of the SUL.

Simple FTP

Purpose SimpleFTP is a java FTP client package which implements some of the functionality that FTP offers [3]. The functions that it supports are the establishment of a connection to the FTP server, the login phase where the user's credentials are being verified, the upload of files as also the change between different directories. More operations, such as delete of a file or download of data are not supported. Therefore, SimpleFTP is a simple package that can be used for the upload of small files to an FTP server.

Client setup - mapper The mapper component which has been implemented for the SimpleFTP Client, is shown in Table 5.5. In this Table, the mapping of abstract inputs to concrete ones is presented in the first and the second column. The third and fourth column of this Table show the mapping of the concrete outputs to abstracted ones. The abstract inputs are evaluated to *valid* or *invalid*, based on weather a user performs a valid or not operation. The input sets for the SimpleFTP are five, since the connect and login is considered as one operation. The "Disconnect" interface is mapped always to an empty set, since it can not be evaluated to any valid/invalid value. If the Learner triggers the "Disconnect" interface, the mapper will

Simple FTP			
Abstract Input	**Concrete Input**	**Abstract Output**	**Concrete Output**
IConnect(Valid)	Valid {host, username, pass}	OK	bool variable = true
IConnect(Invalid)	Invalid {host, username, pass}	IOE	IO Exception returned
IUpload(Valid)	Valid {filename}	NOK	bool variable = false
IUpload(Invalid)	Invalid {filename}		
IDisconnect()	{ }		
IChangeDir(Valid)	Valid {remote directory}		
IChangeDir(InValid)	Invalid {remote directoy}		

Table 5.5: mapper for SimpleFTP

concretise it to the disconnect functionality with an empty set of parameters and the test adapter will call the corresponding function of the SUL, which will not expect any parameters.

In case the Learner for instance selects the abstract input of a valid connect to be sent to the SUL, the mapper will at first receive this action and translate it to a concrete input, which contains the method of the SUL to be called and some valid input parameters. Afterwards, the concrete input is sent to the SUL. We define validity based on whether the information and the data which are sent to the SUL are defined correctly. In other words, a login is valid if the username and the password are specified correctly, a change of the directory is valid, if the given filename exists and the path of the file is correct, etc.

In Table 5.5, the output interfaces for SimpleFTP are also presented. These are generated based on the exceptions thrown by the SimpleFTP implementation. Each time that the SUL is queried, the SUL can response with an "OK" in case of success, with a "Not OK" in case of failure, or with an "Exception" in case of a disruption of the normal flow of instructions. The mapper translates each success to an abstract output OK (OK) string, each failure to an output not OK (NOK) string and each sort of thrown exception to an output abstract String, named each time as the corresponding exception. The SimpleFTP Client thrown only an IO Exception and does not distinguish between other kinds of exceptions.

FTPClient Apache			
Abstract Input	**Concrete Input**	**Abstract Output**	**Concrete Output**
IConnect(Valid)	Valid {host}	OOK	bool variable = true
IConnect(Invalid)	Invalid {host}	ONOK	bool variable = false
ILogin(Valid)	Valid {username, password}	IOE	IO Exception returned
ILogin(Invalid)	Invalid {username, password}	SE	Socket Exception returned
IUpload(Valid)	Valid {filename}	NPE	Null Pointer Exception returned
IUpload(Invalid)	Invalid {filename}		
IDisconnect()	{ }		
IChangeDir(Valid)	Valid {localFileFullName,fileName, hostDir}		
IChangeDir(InValid)	Invalid {localFileFullName,fileName, hostDir}		

Table 5.6: mapper for FTPClient Apache

FTPClient from Apache

Purpose The second client that has been analyzed is the FTPClient class, provided by Apache. It supports all the FTP operations such as directory listing, file upload and download, resume a broken upload/download and much more [26]. It distinguishes between the connect phase and the authentication phase by providing two separate methods; one for connecting to a remote server and one for the user authentication with login.

Client setup - mapper Table 5.6 shows the mapper which was implemented for the FTPClient from Apache. The abstract input interfaces are six, since the connect and the login phase are handles separately by this FTP Client. The abstract inputs are evaluated to valid or invalid, based wether the host address, username, password, filename etc. are correct on not. The output interfaces that the SUL produces, are mapped to "OOK", if the function called has returned "true", to "ONOK", if the corresponding function has returned "false" as also to strings that represent exceptions, thrown by the function implementation. In this FTP Client, the type of the exceptions thrown vary more than in the SimpleFTP and therefore the output interfaces are also more.

FTp4jClient

Purpose The Ftp4j is a library which implements all the features of an FTP client in Java. Transferring files through upload and download methods,

FTP4j Client			
Abstract Input	**Concrete Input**	**Abstract Output**	**Concrete Output**
IConnect(Valid)	Valid {host}	OK	bool variable = true
IConnect(Invalid)	Invalid {host}	NOK	bool variable = false
ILogin(Valid)	Valid {username, password}	SE	Socket Exception returned
ILogin(Invalid)	Invalid {username, password}	ILLE	Illegal State Exception returned
IUpload(Valid)	Valid {filename}	FTPE	FTP Exception returned
IUpload(Invalid)	Invalid {filename}	FTPILLE	FTPI llegal ReplyException returned
IDisconnect()	{ }	DTE	FTP Data Transfer Exception returned
IChangeDir(Valid)	Valid {localFileFullName,fileName, hostDir}	AE	FTP Aborted Exception returned
IChangeDir(InValid)	Invalid {localFileFullName,fileName, hostDir}	FNF	File Not Found Exception returned
		IOE	IO Exception returned

Table 5.7: mapper for FTP4jClient

browsing the remote FTP site, creating, deleting, renaming and moving remote directories and files are only some of them [25]. This library can connect to the remote FTP server through proxy or through a direct TCP/IP connection. For this thesis's purposes a TCP connection with the server has been established.

Client setup - mapper As shown in Table 5.7 , the input interfaces for the FTP4j client are exactly the same as for the FTP client from Apache. The reason for that is that both implementations are offering the option to call individually all the methods that we examine. Regarding the output interfaces, the FTP4j client categorises the exceptions thrown by the program in obviously lots of groups, more than the other two clients do. As shown at 5.7, this client implementation can produce eight different kind of exceptions when calling the functions defined in the input interfaces. In that way the user gets easily informed why the operation could not be handled properly by the SUL. We have mapped each of these exceptions to an abstract output "String", which indicates this information. Same as to the other clients, when the operation succeeded, the result is abstracted to an "OK" String otherwise to a "NOK" String.

5.3 Issues and Tunings

In this section issues that have arisen with the initial experiment setup are discussed as also the actions we performed in order to overcome them.

Initially, the first problem we faced was the fact that LearnLib was not able to terminate the learning process, based on this learning setup and mapper construction we described above. More specifically, the different user's rights have not been properly tested. In particular, when an abstract input occurs with the value *Login(Valid)* then a concrete input is created with the credentials of the user1, user2 or user3. In fact, although the credentials are correct, the ftp server will behave differently based on the user who wants to be authenticated.

Let us assume the following scenario:

The Learner wants to query the SUL for a valid login. The Learner will send initially to the mapper, the abstract input "Connect (Valid)" and the mapper should construct a token with valid credentials and send the concrete set to the SUL. However, the mapper at that time has the possibility to create one of the three valid sets for user authentication: either a login with the correct credentials of user1 or of user2 or of user3. If the mapper, selects randomly a valid concrete set each time an abstract "Connect(Valid)" is received, then the deterministic behavior of an automaton will be violated. Each user authentication results to a different flow of events. If the user1 is authenticated, the listing directory will be different as if the user2 had been authenticated. In that way, if a "Change Directory" would be the next query to be applied to the SUL, it would give different output for the same input "Login(Valid), Change Directory(Valid)". Hence, LearnLib would be not able to construct a hypothesis and a counterexample will be always produced. Indeed, when applying this randomly technique to the mapper no learned model was able to be constructed and the learning was terminated after the repetition of thousand of steps.

In order to address such problems and allow the testing of the different user privileges the mapper component as defined before, had to be changed. Specifically, the Learner sends not only an input message for authentication to the mapper but also which user wants to be authenticated. The mapper is then responsible to translate this request to a concrete message for the corresponding user to be certified.

Practically, the mapping for the input interfaces changes to the following:

- Login(ValidA) for user1

- Login(ValidB) for user2
- Login(ValidC) for user3
- Login(Invalid) for an unauthorised user

Another issue that we have faced, was the different access rights that each virtual user has. When encountering the different user's rights, it should also be taking into consideration the initial setup of these rights, as described in Section 5.2.1. Our setup has not been the same for all the users and as a consequence, the different users rights have also an impact to the authorisation process. More specifically, user3 does have writable rights in the root folder, whereby user1 and user2 have only read rights. When user3 tries to be authenticated, although the user's credentials are correct and a valid login is encountered, the FTP protocol responds with a message *"500 OOPS: vsftpd: refusing to run with writable root inside chroot()"*. Thus, although a valid login occurs in terms of data correctness, the authentication fails due to unsuitable permissions. In other words, a Login(ValidC) for user3 will result to an unsuccessful login (output "NOK"), since the FTP server will respond with such a message as described above. In order to trace down this occasion, the mapper maps the previously described response of the SUL to another abstract output, that we have named "OOPS", instead of the abstract output "NOK", that was mapped before. Thus, when a server produces such a message *"500 OOPS: vsftpd: refusing to run with writable root inside chroot()"* , the mapper will map it to the abstracted output "OOPS". In that way, we distinguish this occasion from the case of an invalid login, where the mapper will return "NOK".

6 Experimental Results and Discussion

"Success is a science. If you have the conditions, you get the result [17]."

[Oscar Wilde]

In this chapter, the results of our experiments are presented. In the first section, the results from the learning of the Heartbeat protocol will be introduced, where in the next section we will focus on the results occurred from the FTP protocol analysis. In Section **??** a discussion of our results will be provided and lastly in Section 6.3 our benchmarking will be presented.

In the introduction we have set some goals that we wanted to achieve through this master thesis. After presenting our experiments and results we state that we have meet these goals. Therefore, we are able to answer the initial questions/goals that we have set in Section 1.4.

The first observation we have made, based on our experiments, applies to both protocols-cases, is related to the mapper component and answers one of the questions that we have defined when starting this theses:

How much influence can have the Learning Setup to the final results?

For both experiments we built a mapper that reduced the number of concrete inputs and outputs to a small set of abstract inputs and outputs. Through our experiments, we found out the importance of the mapper into the learning procedure and its determined role, since it has a really big influence and impact into the learned models. A not so precise implementation of the mapper, can result in divergent final results. Thus, even a small change to the mapper behavior can lead to a dissimilar outcome. The functionality of the

mapper should be only restricted into translating concrete values to abstract ones and vice versa. No additional control behavior should be added, as for example equality checks, checks between previous and next messages etc. It is important that such operations are observed during the learning phase and not having been predefined. Thus, based on the experiments performed, the correctly setup and implementation of the mapper component is a major issue that should be taken into account in advanced.

The next two sections present and discuss our results for the FTP and the Heartbeat protocol and answer the following questions that we have defined when starting with this thesis:

Could possible errors and unexpected system's behavior be uncovered through applied automata learning techniques?

Do the network protocols that are being learned comply to their specifications?

Lastly, the last goal that we have set initially is the following:

How is the performance of the "learning" affected when different equivalence methods are used and which optimisations can we perform?

which is presented and discussed in the *Benchmarking* Section 6.3

6.1 Heartbeat protocol

6.1.1 Results

We applied learning-based techniques to learn the behavior of two implementations (that we have developed) of the Heartbeat protocol; the correct implementation which complies to the specification and a faulty one which includes the Heartbleed bug. Our purpose have been through learning the behavior of the two implementations, to compare the two generated models in order to uncover unexpected behavior, which refers to the Heartbleed bug.

We were able to learn a model consisting of one state, which represents the HeartbeatRequest-HeartbeatResponse exchange. But we could not learn the

complete behavior of the Heartbeat protocol by using the learning setup presented in Section 5.1. As shown in Figures 6.1 and 6.2 when an input is given to the SULSUL, it will produce an output and it will stay always at the same state. Thus, no other states will be created. Different inputs will produce different outputs, but will result to the same state. That is due to the fact, that every message sent is an independent event from the next message that will be sent. In Figure 6.1, the learned state model for the correct implementation is shown, whereby in Figure 6.2, the model of the faulty one is presented. As described before, the Learner sends HeartbeatRequests to the SUL and the SUL answers with a HeartbeatResponse, which" is evaluated to valid or invalid. If a valid HeartbeatRequest is sent, then both implementations -the correct and the faulty one- will respond with a valid HeartbeatResponse. However, if an invalid HeartbeatRequest is sent from the Learner to the faulty implementation of the SUL, then an valid response will be produced, since the SUL will not perform the corresponding checks and despite the fact of an invalid request, it will answer with a valid response. Based on the definition of "Valid response" as defined in Section 5.1, a response is valid when the length of the response equals the length of the request message (not the actual length). On the other hand, when an invalid request is sent to the correct implementation, the SUL will perform the needed checks and will return the message that should be returned and not the message with another length, as defined in the request message. Therefore, the HeartbeatResponse will be evaluated to invalid, since the length of the response does not equals the given length of the request message, but equals the actual length of the message. This is illustrated in Figure 6.1, where in the model produced for an invalid HeartbeatRequest an invalid HeartbeatResponse (the length of the response does not equal the length requested) will be generated.

6.1.2 Discussion

chap:Discussion

LearnLib will learn that behavior of the system but it will produce a model with only one state, since the next set of input HeartbeatRequest/output HeartbeatResponse, will be independent of the previously observed set. For

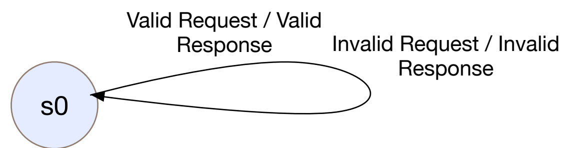

Figure 6.1: Learned Model for Correct Implementation of Heartbeat Protocol

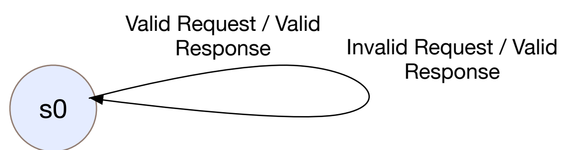

Figure 6.2: Learned Model for Faulty Implementation of Heartbeat Protocol

that reason there is no transition to a next state and thus the model will have only one state. Although the learning was not as we expected, we could state that the comparison of these one-state models (6.1, 6.2), could reveal that for the same input a different output is generated. As a result, since the models are not identical, this could be a basis for a further investigation of this observation.

6.2 FTP Protocol

6.2.1 Results, common in all FTP Implementations

First Result: We performed the L* learning algorithm in order to learn three different implementations of the FTP protocol. We were able to successfully learn all the three FTP client implementations for the FTP protocol, under the condition that user's privileges categorisation has been initiated, as it will be explained below. By "user's privileges categorisation" we mean the issue that we have discussed in chapter 5.3. As stated there, when a *Login(Valid)* was given as an abstract input to LearnLib, then a concrete input was initiated randomly; login credentials either for user1 or user2 or user3. As a result, we have observed that the deterministic behavior of LearnLib did not allow to learn and construct a model with this setup, since each user's authentication results to a different flow of events. Hence, LearnLib has been terminated with an exception, after trying about one hour to construct the model. This result has been observed in all three FTP protocol implementations we examine.

Discussion

This behavior of LearnLib is expected and absolutely correct, since with our initial setup we have tried to learn a non-deterministic automaton; for each state and input, more than one transition was possible. Each time a valid login is performed another user is authenticated randomly, so for the same state and input (Login(Valid)), different transitions were possible.

	User1	User2	User3
SimpleFTP	4-State Model	4-State Model	No Model
FTP4j	4-State Model	4-State Model	No Model
FTPClient Apache	9-State Model	9-State Model	No Model

Table 6.1: Results for the three FTP Client implementations

	1. Input –>Output	2. Input –>Output	3. Input –>Output
SimpleFTP	ConnectLogin(Valid) –>500	**ChangeDir(Invalid) –> IO Exception Or NOK**	
FTP4j	Connect(Valid) –>OK	Login(Valid) –>500	**Login(Valid) –> FTPE Or IO Exception**
FTP Apache	Connect (Valid) –>OK	Login (Valid) –>500	**Login(Invalid) –> IO Exception Or NOK**

Table 6.2: Traces for each implementation

Second Result: In Table 6.1, the resulted learned models for each user at each FTP implementation are presented. As it can be observed, for "user3", no model was able to be generated in all of the three FTP implementations.

As described in Chapter 5.2.1, the virtual user3 has rights to upload and download files, under the users home directory: /var/www/user3. However, this directory is not set to "read only". This causes the vsftp server to refuse to authenticate and grand access for user3 to his directory, by producing the following message: "500 OOPS: vsftpd: refusing to run with writable root inside chroot()". The fact that user3 has write and read privileges for his root folder is not permitted from vsftp server and therefore the server does not allow the user to login.

Discussion

The reason why no model could be created for "user3" lies to the fact of a non-deterministic behavior of the vsftp server. When the "user3" tries to be authenticated, the vsftp server sometimes resets the connection and some other times not. That means that sometimes the user is disconnected (in case of connection reset) and sometimes the user stays connected but can not be authenticated (in case of not connection reset). This behavior of the

vsftp server has been also tested from the terminal and indeed the responses do not always coincide, although the given commands where always the same. Therefore, also without any FTP Client implementation to be used but only by using the terminal, the vsftp server behaved randomly when trying to authenticate "user3". Therefore, the reason why no model was able to be created lies to the vsftp server and not to the different FTP client implementations.

Table 6.2 shows how the three FTP implementations reacted to this arbitrary server behavior, when user3 performed a login action. In particular, the columns of the table show a trace (1.Input → 2.Input → 3.Input), where it was first identified at each FTP implementation this random server behavior. Thus, when observing each row of the table, we receive information regarding which input has triggered which output at each FTP under learn. The output "500" that is stated in the table stands for the server reply code "500" and more specifically for the produced message: *500 OOPS: vsftpd: refusing to run with writable root inside chroot().* With yellow, are marked in the Table 6.2 the outputs, which have cause a non-deterministic behavior of the system and therefore LearnLib has terminated the learning procedure with an exception.

In particular, for SimpleFTP when the input trace *ConnectLogin(Valid)*, $ChangeDir(Invalid)$ was given for the user3, the SUL produced for the last input either NOK as output or has thrown an exception (IOE). The same output is also produced from FTPClient from Apache, for the trace $Connect(Valid)$, $Login(Valid)$, $Login(Invalid)$ either an exception or a "NOK" was returned. In FTP4j Client, for the trace $Connect(Valid)$, $Login(Valid)$, $Login(Valid)$ the SUL responded to the last input with either $FTPE$ or IOE. The IOE exception is produced when the server (vsftp) has reseted the socket connection and the ftp socket therefore is initiated to null. When the server has not reset the connection, then the output NOK is produced in case of SimpleFTP and Apache FTP. These two FTP implementations produce NOK, since they are calling their login method, which returns a boolean parameter corresponding to whether a successful login has been performed or not. In case of FTP4j client, the login method for user3 produces an "FTP Exception", when the reply code from the server is not "230" (User logged in) or to "332" (Need account for login). Since in this occasion the reply code is "500", then an FTP exception is returned from the code.

As a last remark, we have tested additional this result by creating some test cases and let them run over a loop of thirty iterations. What was observed was the same result; the server produced sometimes a connection reset and sometimes it just did not authenticate the user3. The tests have been executed several times and no pattern was able to be identified. The reaction of the system was uncontrolled and arbitrary. Thus, by using active learning techniques we were able to make such an observation, which maybe would not have been detected so easily when performing other testing methods.

Notice

In the next sections, the results for each FTP Client implementation are seperately presented and discussed. We show and discuss the results, based on our second setup where the issue with the *random user selection* does not occur anymore. As explained in Chapter 5.3 the issue has been solved by adding additional information in the input interfaces, which indicates which user asks for authentication each time. Moreover, we present in the next sections the learned models only for "user1" for the following reasons:

- No learned models were generated for "user3" due to rights issues as described in the above section.
- The learned models for "user2" result to the same model as for "user1", so for simplicity reasons we have chosen to present the learned models of one user.

The original models, as generated from LearnLib are attached to *Appendix*. In the next sections, we present the learned models in our own graphs (Figures 6.3, 6.4, 6.5), which have exactly the same states and transactions as the original ones as attached in *Appendix*. The reason why we have used our own visualisation is only because the original graphs are not understandable at the first sight, since they contain lots of transaction lines that we have merged to a minimal number.

6.2.2 SimpleFTP

Results

In Figure 6.3 the resulted learned model from SimpleFTP Client is presented. SimpleFTP is expected to have one state less than the other two FTP clients, because the connect and login phase of this client are encountered as one method. For that reason the state "Connect" is interpreted as "Connect and Login" for the SimpleFTP Client. This SimpleFTP client is successfully learned from LearnLib and the learned model for this client consists of four states as shown in 6.3, that we can name as following:

1. Disconnected (s0)
2. Connected and logged in (s1)
3. Directory Changed (s3)
4. Connected but not logged in (s2)

In Figure 6.3 the learned model is shown. The model infers to a Mealy Machine of the system's implementation that has one input and one output. Each transition edge is labeled with the value of the input and the value of the output. Moreover, from every state all possible inputs result to a transition. When a transition results to the same state it is illustrated in the figure 6.3 with a loop arrow. For simplicity reasons, they are not pictured the input and output sets when a loop error appears. In this case, it is implied that in the loop arrow all the other inputs/output sets are executed.

Discussion

As we can see at Figure 6.3 the initial state of the learned model is the state "Disconnected", which means the user is not connected and not logged in (ftp socket is null). From this state, all the operations except the "ConnectValid)", such as change directory, upload etc. will output an IOException. The second state shows that the user has been connected and logged in, where the third state indicates a successfully change of the current directory. The last state, is the state where the user is connected but not logged in and is a different state from the state "Disconnected", because the ftp socket is initiated with

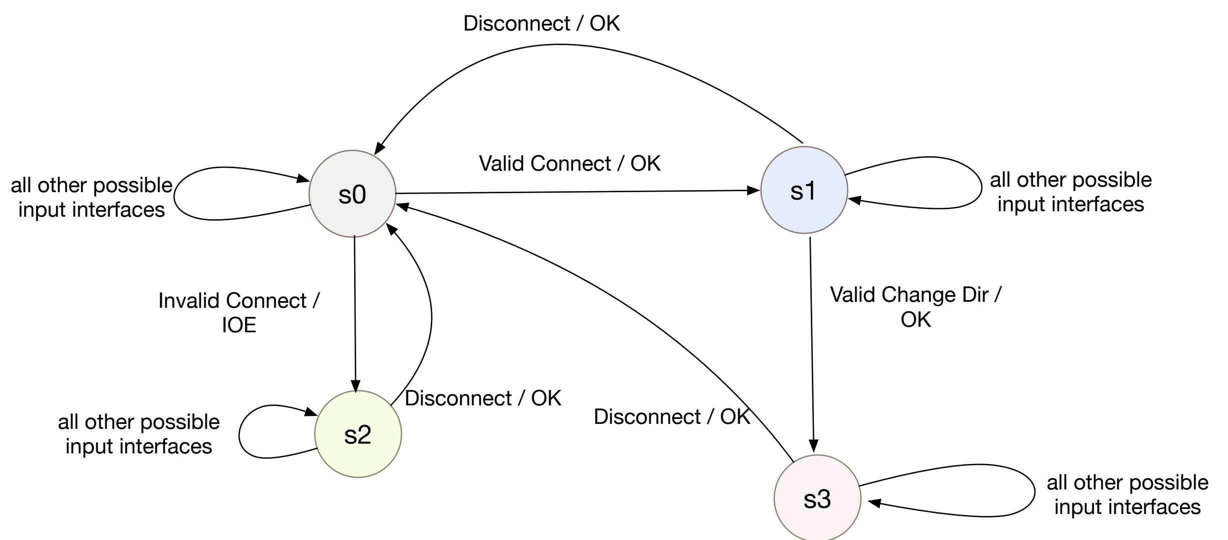

Figure 6.3: Learned Model for Simple FTP Client

some values. Therefore, when an operation, such as change of directory, is performed, no IO Exception will be now produced because the ftp socket is not anymore null.

The model complies to the FTP protocol specification although is a very simple FTP client implementation with some limitations when using it.

6.2.3 FTP4j Client

Results

In Figure 6.4 the learned model for the FTP4j Client is presented, with the input/output interfaces as described in Section 5.2.2. Thus, the FTP4j Client has been learned successfully, since the learning algorithm has been terminated and has also produced a learned model.

The learned model of the FTP4j Client consists of four states that we can name as follows:

1. Disconnected (s0)
2. Connected (s1)
3. LoggedIn (s2)
4. Directory Changed (s3)

As we can observe from the Figure 6.4 when an “InvalidConnect” or a “Disconnect” is given as input to the system, the SUL will move always to state *s0*, which is the *Disconnected state*. When an “InvalidLogin” or a “Valid Connect” is given as input and the SUL is already “Connected”, then it will move to state *s1*, which is the “Connected” state.

Discussion

We found out that the FTP4j client does not comply with the FTP protocol’s specification as defined in RFC[66]. According to RFC specification, *“The control connection shall be closed by the server at the user’s request after all transfers*

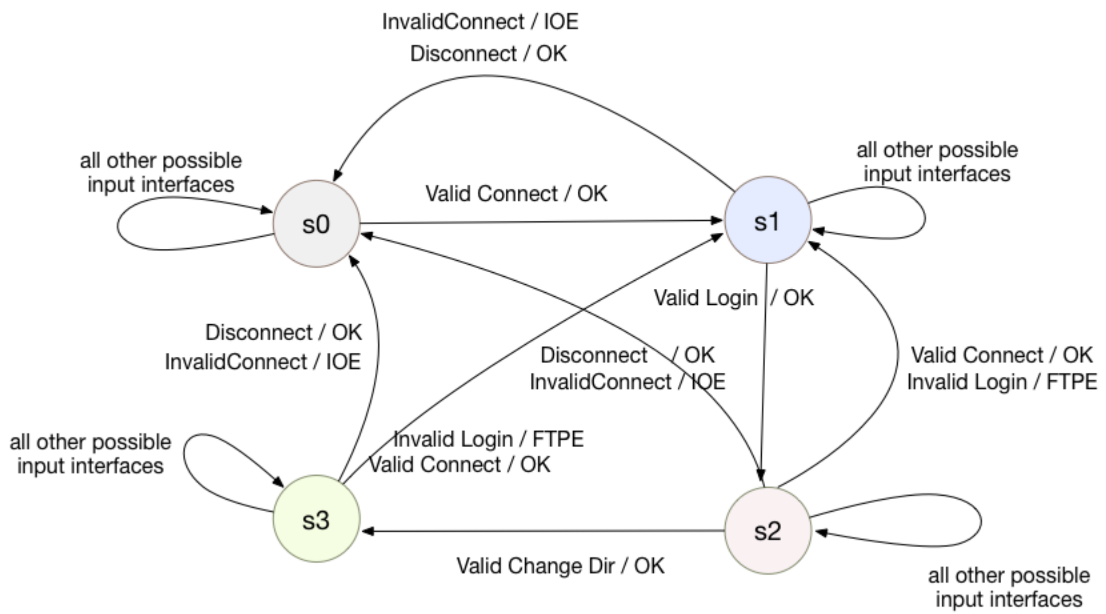

Figure 6.4: Learned Model for FTP4j Client

and replies are completed. An unexpected close on the control connection will cause the server to take the effective action of an abort (ABOR) and a logout (QUIT)."

On the contrary to this specification, the FTP4j Client will behave as follows: If an invalid connect is received after the user has been logged in, then the FTP4j Client will respond with an IO Exception but it will react like a "Disconnect" operation; the user will be disconnected from the server, although this action has not been triggered. If afterwards, a "Disconnect" request is performed, the FTP4j client will respond with an exception, since the user was previously disconnected when an invalid login was sent. This behavior is illustrated in Figure 6.4, when we are for example in state *s1* and we give as input to the SUL an "InvalidConnect", then it will move to state *s0*, which is the "Disconnected" state. Thus, the FTP4j client, in case of an invalid connect, will produce a socket exception and it will acts like the SUL is disconnected.

The specification says that the control connection shall be closed by the server at the user's request. In the case described above, the user (client) does not request to close the connection, but the connection is closed without the client's demand. We have performed the same actions/steps when connecting to the FTP server from the terminal, where no FTP client implementation is used. In this case, the FTP server did not close the connection when we sent an invalid connect from the terminal as an already logged in user. Thus, we conclude that this faulty behavior results from the FTP4j client, which does not comply to the protocol's specifications. Therefore, by applying automata learning techniques, we were able to uncover an incorrect and non-compliant behavior of the FTP4j Client. However, our experiments did not cover all the possible operations that can be performed from the FTP protocol, but only a small amount of them. Maybe with learning methods could be uncovered more faults in the FTP implementations, if more operations of the FTP protocol are tested.

6.2.4 FTPClient Apache

Results

Lastly, the model learned from FTPApache Client is shown in Figure 6.5. For simplicity reasons, they are not pictured the input and output sets when a loop error appears. In this case, it is implied that in the loop arrow all the other inputs/output sets are executed.

As we can observe, this learned model is much more complex in comparison to the previous ones. This is due to the fact that some paths of this model show a misbehavior in respect to protocol's specification, that will be discussed in the "Discussion" section below.

LearnLib was able to learn the behavior of the FTPApache Client and construct a model. The model constructed consists of 9 states but to some of them (states *s5,s6, s7,s8)* we can not give a name, as we performed for the other learned models, since they do not represent any logic situation. States *s0-s4* we can name as follows:

1. Disconnected (s0)
2. FTPNotNull (s1)
3. Connected (s2)
4. LoggedIn (s3)
5. Directory Changed (s4)

The state *s1* is named "FTPNotNull", because it is an intermediate state before we are logged in to the FTP server. That means, if we are disconnected (state *s0* and we perform an invalid connect, the Apache FTP Client will produce a socket timeout exception (output "SE"). Thus, the client tries to connect to the server but since the given connect data are invalid, the connect will fail but we will not be disconnected. Therefore, a new intermediate state is created since we are not directed back to the "Disconnected" *s0* state. We name this intermediate *s1* state "FTPNotNull", because the ftp object is not null anymore as when we are disconnected but on the other hand we are also not connected to the server. This behavior complies to the FTP specification defined in RFC[66], as has been described in the previous Section 6.2.3 where the FTP4j Client did not complied to it: *"The*

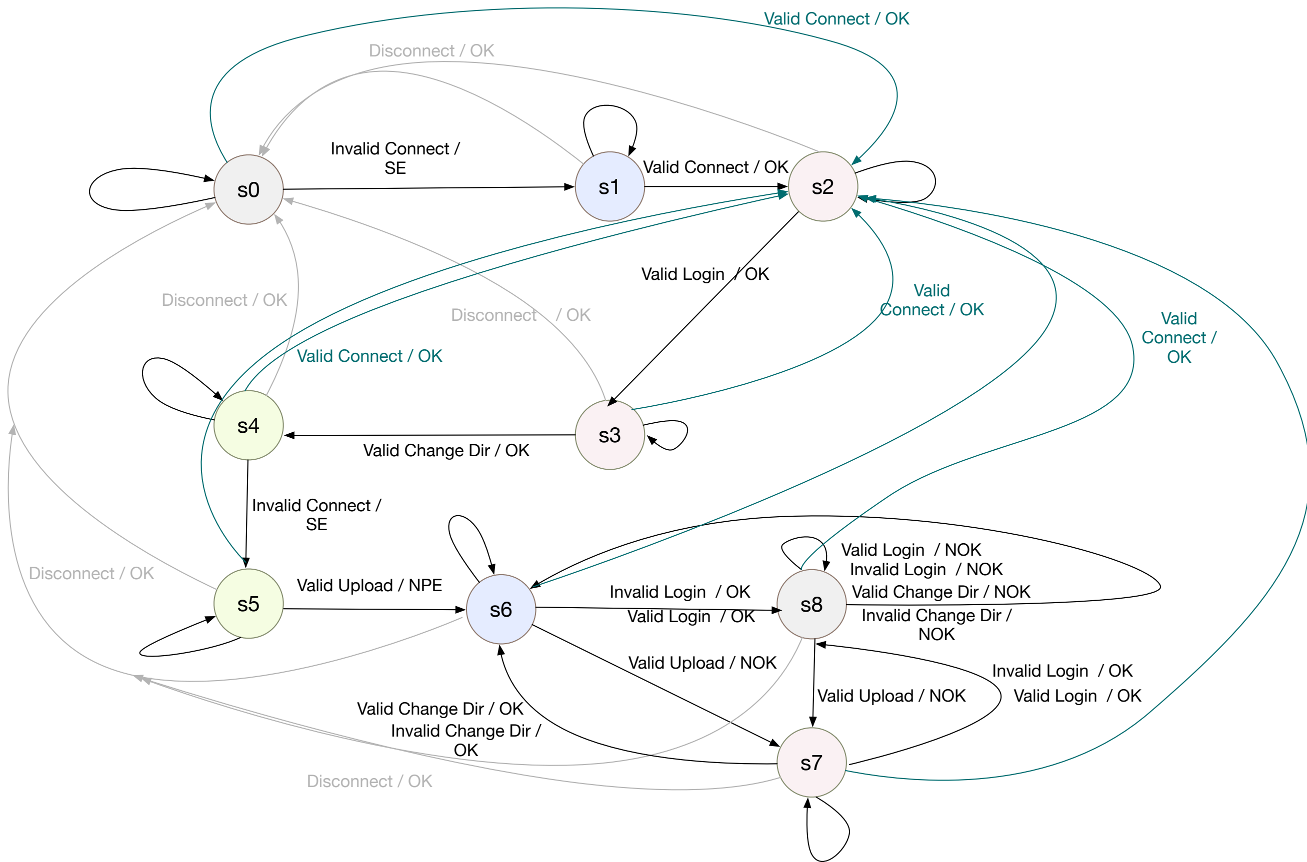

Figure 6.5: Learned Model for FTP Client Apache

control connection (Disconnect) should be closed only at the user's request". The difference between the *s0* state and the *s1* state is that when we send an invalid connect to the *s1* state the client will respond with the output "NOK" and not with a Socket Exception (output "SE") as in the state *s0*, because now the socket is initiated and it is not null. This is why we have these two similar but not equal states.

The states *s2, s3, s4* as shown in the learned model 6.5 are common as in the other FTP implementations.

Discussion

What is remarkable from the learned model, are the states *s5, s6, s7, s8* which do not make any sense and reveal a faulty behavior of the Apache FTP implementation.

For instance, if we start from state *s1* and follow this trace as shown at Figure 6.6:

$ValidConnect, ValidLogin, ValidChangeDir, ValidConnect, ValidUpload, Invalid$ $Login$

we will move to states:

$s1, s2, s3, s4, s5, s6, s8$

As we can observe from this trace, for an invalid login, the FTPClient will respond with "OK", where a "NOK" should have been produced. However, for every invalid login, given as input before the state *s5*, the SUL will respond correctly with a "NOK". After state *s5*, some other misbehaviors also occur, as for instance the input/output $ChangeDr(Invalid)/OK$ when moving from state *s7* to state *s6*. Moreover when we are in state *s7* and an invalid login is given as input, the SUL will respond with "OK". However, from the same state *s7* if a valid connect is performed, as indicated with blue color in Figure 6.5, the SUL transits to state *s2* and starts behaving correctly again till state *s5*.

We have tried to exploit this misbehavior of the FTP Client by trying to authenticate an invalid user after an invalid connect has occurred. Although

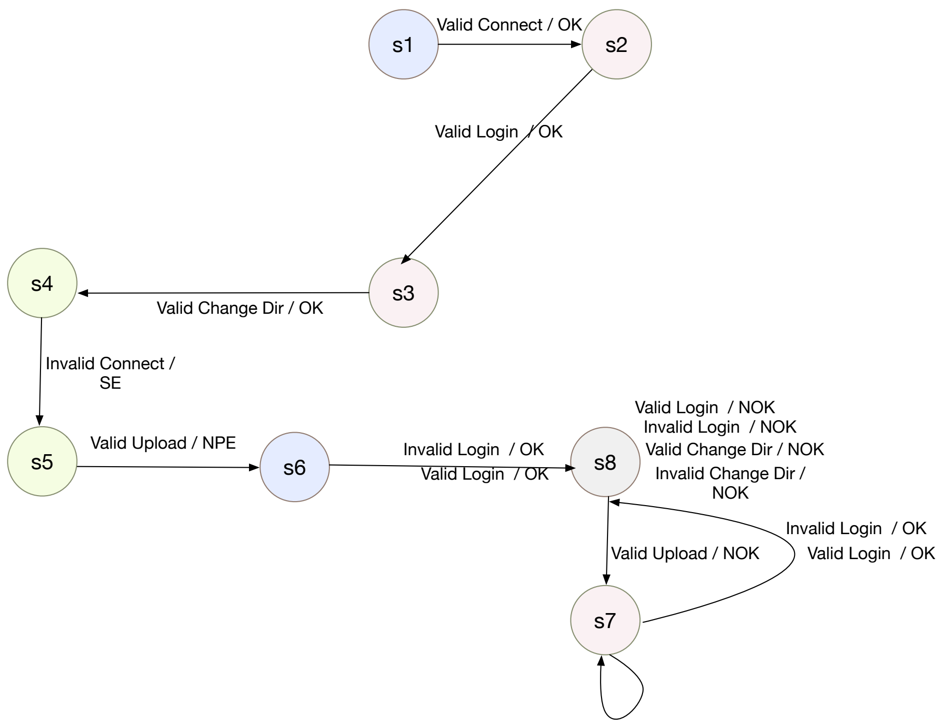

Figure 6.6: Example Trace from FTP Client Apache

SUT	Memb.Queries	Total Running Time	Total time of Eq.Queries	Total time of Memb.Queries	States Learned
SimpleFTP	232	1m 22s 502ms	87ms	1m 22s 415ms	4
Apache FTP Client	640	14m 23s 219ms	2s 28ms	14m 21s 191ms	9
FTP4jClient	370	12m 40s 481ms	10s 13ms	12m 30s 468ms	4

Table 6.3: Statistic Results from learning the FTP Protocol using Random Walks

SUT	Memb.Queries	Total Running Time	Total time of Eq.Queries	Total time of Memb.Queries	States Learned
SimpleFTP	232	1m 23s 12ms	42ms	1m 22s 970ms	4
Apache FTP Client	640	14m 20s 805ms	32ms	14m 20s 773ms	9
FTP4jClient	370	13m 25s 83ms	23ms	13m 25s 52ms	4

Table 6.4: Statistic Results from learning the FTP Protocol using W-Method

the client returned "OK' for an invalid login, no data could be retrieved or uploaded for this user. However, we reported this misbehavior of FTP client to Apache developers and they have stated that the issue will be fixed in newest versions [41].

6.3 Benchmarking

When learning a system, what is also important except of the final results, is the overall execution time as also other statistical data as for example number of membership queries, number of equivalence queries, etc. In that way, we can understand which actions are time consuming and therefore optimisations can be performed. Thus, we have collected some benchmarks for the different FTP client implementations that we have learned and we will present in this section.

The experiments have been implemented in a virtual machine and they have been executed several times and the average execution time for each implementation has been calculated. In that way, the results are stable and can give a good overview of benchmarking. We have collected some statistics in the three different implementations of the FTP protocol as presented in Tables 6.3, 6.3 by using two different equivalence methods: the Random Walks and the W-method as described in Section 3.1.

As observed, in the SimpleFTP, the total time of equivalence queries was reduced to 50% when using the W-method. The total running time as also the time of membership queries was quite the same for both methods.

Learned states as also the number of membership queries has been the same in both methods for all FTP implementations. Regarding the FTP client from Apache, the total execution time has been reduced by three seconds, when applying the W-method, where the time of membership queries has been almost the same in both methods. The equivalence queries execution time however, has been reduced by 70%, when using the W-method. The time of equivalence queries for the FTP4j client for W-method is 440% times faster as with Random Walks. Hence, as we observe, by using the W-method the time of equivalence queries can be extremely reduced although the overall execution time is not being that affected.

During automata learning, depending on the target system's size, several membership queries may be produced for construction of the final hypothesis. Each of these queries consists of the actual execution time of the system under learn and the execution time of the reset action [56]. A reset action is always needed between each membership query, since the queries should be independent from each other. As shown in our benchmark results, the time the membership queries need is almost 99% of the overall execution time. Therefore, a focus should be given to improve this part of learning. Applying for instance application-specific filters, which are answering queries by using preexisting knowledge, would eliminate the time effort, since the SUL is not needed to be asked directly. Query cashes which answer queries from responses of previous queries or also test drivers, which instrumenting the SUL [56] are also methods that can be used for optimisation within LearnLib.

7 Related Work

"Unity is strength, when there is teamwork and collaboration wonderful things can be achieved [75]."

[Mattie J.T Stepanek]

Automata learning and learning-based testing are popular techniques that have been applied in several cases in order to test security vulnerabilities [4]. A number of attempts have also been made for automatically extracting system's specifications by observing its behavior [58]. Moreover, lot of effort and focus has been given in the field of active learning and modelling of systems. Therefore, this section will explore in more detail the context of projects from other researchers, related to the field of learning-based testing and automata learning.

Learning automata has been analysed and discussed by a lot of researchers in different ways. In particular Steffen, Hower et al. have demonstrated in their paper [70] a scenario of active learning of black-box systems by modifying the L* learning algorithm for Mealy machines. Moreover, in the paper of Kumpati Narendra [58] a survey has been conducted in the area of automata learning for analysing the behavior of learning automata, discussing the interaction between several automata as also issues that may arise when working with automata. In this master thesis we explore the field of automata learning. However, whereas the aforementioned papers focus on analysing the current use and behavior of learning automata as also on experimenting with new methods that can be fruitful in this field, we focus on learning the system's behavior by applying learning automata techniques. In their paper, Nowe, Verbeeck, et al. have also summarised some theoretical results from the field of Learning Automata to show that learning automata can be efficiently used as basis to construct so called 'multi agent' learning algorithms [60]. This work also explores a more theoretical aspect

of automata learning and does not focus on the application of them in real systems, as demonstrated in our work.

Based on our research throughout lots of papers in the field of automata learning, we concluded that one of the most widely used learning algorithm used in the field of automata learning is the L* algorithm. First Angluin has demonstrated in her paper [9], as stated, a way to identify an unknown regular set from examples of its members. In particular, she exhibits an algorithm - L* algorithm- for learning any regular set presented by *a minimal adequate Teacher* and can indicate if it is equal to the unknown set or not. In our work we have used this algorithm as our basic learning algorithm for learning a black box implementation of the system under test each time. The L* algorithm is in detail presented in Section 2.1.1. A different approach is proposed by K.Meinke and M.A. Sindhu in their paper [55], were they introduce an incremental learning algorithm, called *Incremental Kripke Learning (IKL)*, for learning and modelling reactive systems. The main characteristic of the systems they analyzed, was the fact that they could be modelled as Kripke structures. As they state, if this algorithm is combined together with a temporal logic model checker, such as NuSMV, the result would be an effective learning-based architecture for reactive systems [55]. In this thesis, the reactive systems that are being analyzed are not modelled as Kripke structures but as Mealy machines. We learned the system under test using the L* algorithm from Angluin and we are focusing on the system's specifications.

While the field of automata learning has been explored by many researchers, a more specific focus has been given on testing implementations of network protocols by using such techniques. Since the systems that we examine in this thesis are network protocols, we give a short sight in the work of other researchers in this area. More specifically, Joeri de Ruiter and Erik Poll have tested in [68] different implementations of the Transport Layer Security (TLS) protocol. However, in our work we focus only in one extension of the TLS protocol, the Heartbeat extension. The implementations that they examine, have been tested by an automated and systematic analysis called *protocol state fuzzing*. Their goal has been to detect security flaws and misbehaviours actions of the TLS protocol. As they stated, the Heartbleed bug of the Heartbeat protocol was not able to be uncovered using this technique. In our results, we could also not reveal the Heartbleed vulnerability with

the learning setup that we have used for learning the Heartbeat protocol. Nonetheless, their learning setup is based on using automated learning techniques to automate the states resulting from the system's implementation and afterwards fuzzing different message sequences. For modelling the protocol's behavior, they have used LearnLib, a learning framework which is also used in this thesis.

Another interesting attempt for automated extraction of protocol specifications for "smart" fuzzers has been made by Serge Gorbunov and Arnold Rosenbloom, as demonstrated in their paper [30]. As they describe, they have introduced AutoFuzz, a *"man-in-th-middle, semi deterministic network protocol fuzzing framework"*, for testing implementations of network protocols. In particular, their focus has been on fuzzing the server side of the File Transfer Protocol (FTP) implementation. In our work we also examine different implementations of the FTP protocol by applying only learning methods to them but we are focusing on analysing the Client FTP implementations. On the contrary, they have focused on the fuzzing framework, used for discovering flaws in the software, by observing its behavior when a specific parameter is given as input. Another attempt to test protocol implementations has been performed by Fides Aarts, Harco Kuppens et al [6], who showed how active learning can ensure the correctness of a protocol implementation when a reference implementation R of it is given. They have used learning tools to reveal errors in the implementation of the bounded retransmission protocol. In particular, they have presented an approach in order to establish conformance of implementations, by using active learning techniques for learning a model M(R) of a reference implementation R. Learnlib and Tomte, are the tools that they have used for the active learning. In our work, we have used LearnLib as learning tool but we have focused on 'learning' rather on 'testing' of the network protocols.

Some more applications of automata learning that have been deployed in the fields of security and network protocols and are mostly relevant to our work are presented below. As presented in their paper [4], Fide Aarts, Joeri de Ruiter et al. have applied learning methods to the software of banking smartcards for evaluating the security of these systems. Unexpected functionality, missing system behavior, mistakes in the design and the implementation are only a few incidents that can be uncovered by learning techniques. Their main focus targets on assuring the secureness and correctness of the system

and in particular of the smartcards used in banks. In our work we also focus on the correctness of the systems under test with respect to a given specification. The software they are testing is considered to be a black-box implementation and therefore they have constructed models based just on observations of the smartcards's behavior; same approach as also used in this thesis. In addition, they have used Mealy machines for modelling the reaction of smartcards, as also used in this thesis. As they stated, by applying the Angluin's L* algorithm to authentic bank cards, they were able to obtain differences between the generated models of the bank cards. The approach of learning network protocols has been also explored from Paul Fiterau-Brostean, Ramon Janssen et al. at their paper [20], where they have explained how the TCP Network protocol can be learned based on observations. They have analyzed different implementations of the protocol in two different operating systems; Windows 8 and Ubuntu. From their analysis they were able to extract that the behavior of the protocol in both platforms is non-compliant to its specification. Our approach for learning protocols is very similar to their work. They have used Angluin's L* algorithm which is implemented in LearnLib. Thus, LearnLib and Tomte, a tool described in Section 3.3, have been their main Java-based learning tools and Wireshark[84] has been used for monitoring the network traffic between the client and the server. The L* algorithm has been also our core learning technique for learning network protocols. Their basic setup included a test adapter, like we have used and described in Section **??**, which is placed on the client side. Moreover, they have used a mapper, as we have also used in our learning setup as described in Section 3.1.2, which is placed in-between the learner (LearnLib) and the adapter, which is connected to the server. In particular, based on this mapper they have abstracted away messages sent to the client and they have concretise messages sent to the server. As they stated in their final results, different transitions between the state machines generated on each operating system indicated a different behavior of the protocol. This learned behavior did not follow the protocol's standard.

As described above, the field of learning automata as also testing implementations of network protocols with learning techniques has been explored by a lot of researchers and as shown in their papers great results have been achieved. Hence, related work efforts as also the experiments of this thesis indicate that automata learning is a fast growing testing method, which

can be very effective and important not only for understanding the systems under test, but also for conformance testing.

8 Conclusions and Future Work

" There no limits to what science can explore."

[Ernest Solvay]

In this thesis we have discussed and applied active automata learning techniques for learning network protocols. The network protocols that we have examined and tried to learn are the FTP protocol and the Heartbeat protocol. Hence, by applying learning methods, we were able to learn models for the FTP protocol. For the Heartbeat protocol a model with only one state has been generated. Although this model does not "learn" the actual behavior of the Heartbeat protocol, it gives information which could be used as basis for further investigation. By using abstraction techniques, we were able to reduce the number of concrete inputs and outputs to a smaller set and abstract away some irrelevant to our purpose information. We run our experiments on FTP protocol, by using the same input alphabet for all the FTP client implementations each time and also we apply the active automata learning method to the Heartbeat protocol. Regarding the FTP protocol, we have analysed three different client implementations in respect to the protocol's specification: the FTP4j Client, the FTP client from Apache and the SimpleFTP Client. We demonstrate that the FTP4j Client does not comply to the FTP specifications, as defined in RFC standard, whereby the FTP client from Apache misbehaves after an execution of a certain trace. In addition, the FTP server used for our experiments behaves in a nondeterministic way, when a user with certain rights tries to be authenticated and therefore LearnLib was not able to generate any model in that case.

We run our experiments by using two different algorithms for equivalence checking during the learning phase; the *Random walks* method and the *W-method*. By collecting some benchmarks information during learning, we

showed that the W-method can reduce a lot the equivalence-checking time, in comparison to the Random walks method. However, 99% of the total running time consume the membership queries. Therefore, we encounter in the future to use optimisation methods for reducing the time of the learned models. Such methods, can already be used within LearnLib, such as application-specific filters, which are answering queries by using preexisting knowledge. Query cashes which answer queries from responses of previous queries will be also taken into account in our future work, since then we will not need to ask queries the SUL directly and the overall running time can be reduced. Another long term goal is the automation of the learning process through an automation of the mapper component. Algorithms can be used for constructing automatically the mapper component and eliminate the developer's effort. Lastly, we consider in our future work the construction of a different learning setup, for learning the behavior of the Heartbeat protocol. In particular, in our current learning setup, the Learner plays the role of the client which generates Heartbeat requests and the SUL plays the role of the server, which answers with Heartbeat responses. However, the learning setup can be separated into two parts; one which focuses on learning the behavior of the client and another which would learn the behavior of the server. In that way, maybe the Heartbeat vulnerability could be uncovered.

Through this thesis and based on related efforts as reference, we show that automata learning techniques can be an effective and fast way to testify the behavior of a system against its specifications.The main advantage of learning is that, by sending messages in an unexpected order we get a high coverage of the code, which is different from for example full branch code coverage, as we trigger many different paths through the code. In addition, inferring systems to learning models is important not only for understanding these systems, but also for model checking and model based testing, which can be performed in later steps.

Appendix

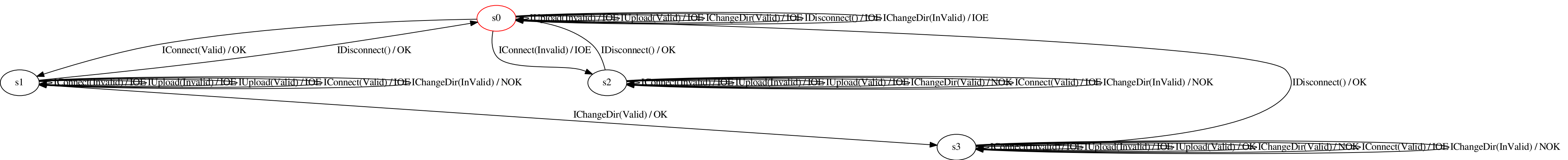

Figure .1: Learned Model from SimpleFTP Client

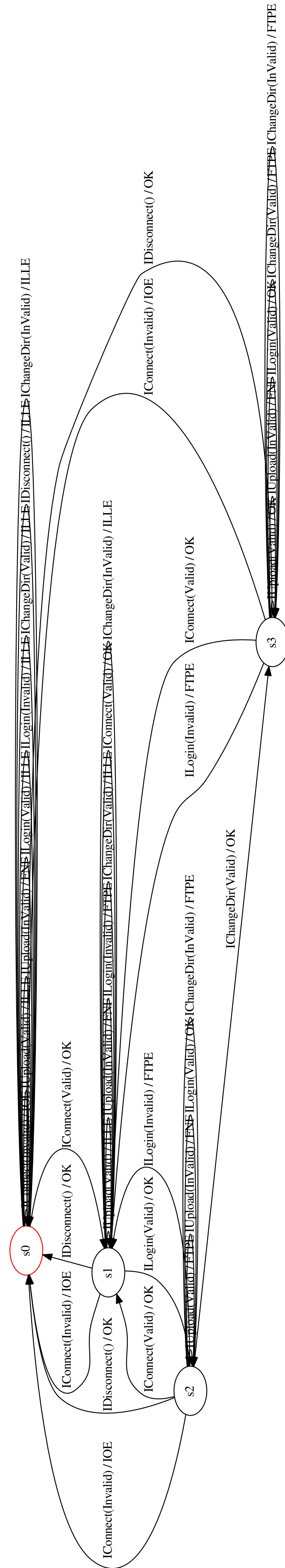

Figure .2: Learned Model from FTP4j Client

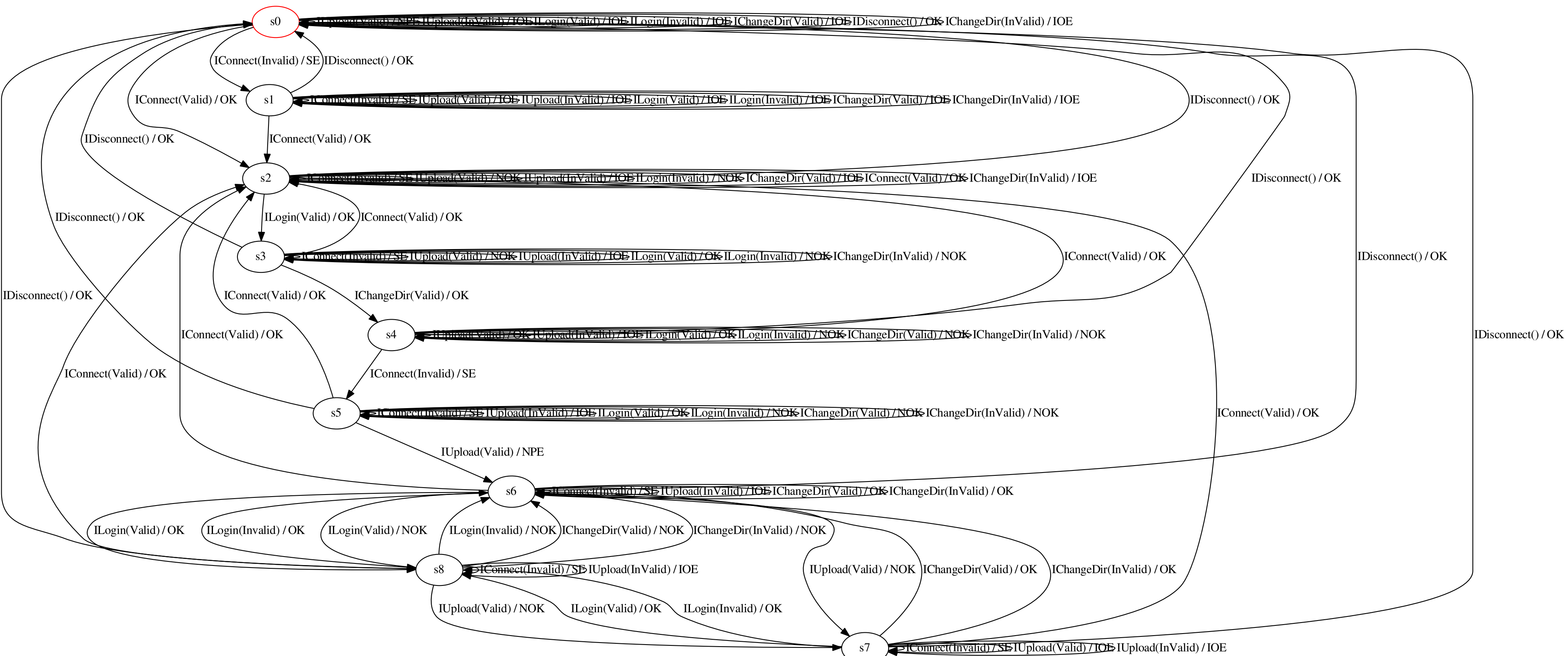

Figure .3: Learned Model from FTP Client Apache

Bibliography

[1] *Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS) Heartbeat Extension, Formal Description from RFC 6520*. URL: https://tools.ietf.org/html/rfc6520.

[2] Publisher: Oxford University Press Published online: 2015 Current Online Version: 2015 DOI: 10.1093/acref/9780191804144.001.0001 eISBN: 9780191804144. *Oxford Essential Quotations*.

[3] *A Simple FTP Implementation*. URL: http://www.jibble.org/simpleftp/.

[4] Fides Aarts, Joeri de Ruiter, and Erik Poll. "Formal Models of Bank Cards for Free." In: *Sixth IEEE International Conference on Software Testing, Verification and Validation, ICST 2013 Workshops Proceedings, Luxembourg, Luxembourg, March 18-22, 2013*. 2013, pp. 461–468. DOI: 10.1109/ICSTW.2013.60. URL: http://dx.doi.org/10.1109/ICSTW.2013.60.

[5] Fides Aarts et al. "Generating models of infinite-state communication protocols using regular inference with abstraction." In: *Formal Methods in System Design* 46.1 (2015), pp. 1–41. DOI: 10.1007/s10703-014-0216-x. URL: http://dx.doi.org/10.1007/s10703-014-0216-x.

[6] Fides Aarts et al. "Improving active Mealy machine learning for protocol conformance testing." In: *Machine Learning* 96.1-2 (2014), pp. 189–224. DOI: 10.1007/s10994-013-5405-0. URL: http://dx.doi.org/10.1007/s10994-013-5405-0.

[7] Fides Aarts et al. "Learning Register Automata with Fresh Value Generation." In: *Theoretical Aspects of Computing - ICTAC 2015 - 12th International Colloquium Cali, Colombia, October 29-31, 2015, Proceedings*. 2015, pp. 165–183. DOI: 10.1007/978-3-319-25150-9_11. URL: http://dx.doi.org/10.1007/978-3-319-25150-9_11.

[8] *An introduction to Machine Learning, University of Oslo*. URL: http://folk.uio.no/plison/pdfs/talks/machinelearning.pdf.

[9] Dana Angluin. "Learning Regular Sets from Queries and Counterexamples." In: *Inf. Comput.* 75.2 (1987), pp. 87–106. DOI: 10.1016/0890-5401(87)90052-6. URL: http://dx.doi.org/10.1016/0890-5401(87)90052-6.

[10] Dana Angluin. "Queries and Concept Learning." In: *Machine Learning* 2.4 (1987), pp. 319–342. DOI: 10.1007/BF00116828. URL: http://dx.doi.org/10.1007/BF00116828.

[11] Keith A. Bartlett, Roger A. Scantlebury, and Peter T. Wilkinson. "A note on reliable full-duplex transmission over half-duplex links." In: *Commun. ACM* 12.5 (1969), pp. 260–261. DOI: 10.1145/362946.362970. URL: http://doi.acm.org/10.1145/362946.362970.

[12] Therese Berg et al. "Insights to Angluin's Learning." In: *Electr. Notes Theor. Comput. Sci.* 118 (2005), pp. 3–18. DOI: 10.1016/j.entcs.2004.12.015. URL: http://dx.doi.org/10.1016/j.entcs.2004.12.015.

[13] Georg Chalupar et al. "Automated Reverse Engineering using Lego®." In: *8th USENIX Workshop on Offensive Technologies, WOOT '14, San Diego, CA, USA, August 19, 2014*. 2014. URL: https://www.usenix.org/conference/woot14/workshop-program/presentation/chalupar.

[14] Dennis de Champeaux et al. "Structured Analysis and Object Oriented Analysis (Panel)." In: *Conference on Object-Oriented Programming Systems, Languages, and Applications / European Conference on Object-Oriented Programming (OOPSLA/ECOOP), Ottawa, Canada, October 21-25, 1990, Proceedings*. 1990, pp. 135–139. DOI: 10.1145/97945.97962. URL: http://doi.acm.org/10.1145/97945.97962.

[15] Yingke Chen. *Machine Learning-based Verification: Informed Learning of Probabilistic System Models, PhD thesis. Department of Computer Science, Aalborg University, 2013.*

[16] Maximilian X Czerny. "Learning-based software testing : Evaluation of Angluin's L* algorithm and Adaptions in practice." In: January (2014).

[17] Joseph M. Demakis. *The Ultimate Book of Quotations*.

[18] Zakir Durumeric et al. "The Matter of Heartbleed." In: *Proceedings of the 2014 Internet Measurement Conference, IMC 2014, Vancouver, BC, Canada, November 5-7, 2014*. 2014, pp. 475–488. DOI: 10.1145/2663716.2663755. URL: http://doi.acm.org/10.1145/2663716.2663755.

[19] *File Transfer Protocol (FTP)*. URL: https://www.ietf.org/rfc/rfc959.txt.

[20] Paul Fiterau-Brostean, Ramon Janssen, and Frits W. Vaandrager. "Learning Fragments of the TCP Network Protocol." In: *Formal Methods for Industrial Critical Systems - 19th International Conference, FMICS 2014, Florence, Italy, September 11-12, 2014. Proceedings*. 2014, pp. 78–93. DOI: 10.1007/978-3-319-10702-8_6. URL: http://dx.doi.org/10.1007/978-3-319-10702-8_6.

[21] Militon Frent. "Correctness: a very important quality factor in programming." In: 1 (2005), pp. 11–20.

[22] *FTP Bounce Attack*. URL: https://tools.ietf.org/html/rfc2577.

[23] *FTP Protocol, formal description from RFC: 765*. URL: https://www.ietf.org/rfc/rfc959.txt.

[24] *FTP Protocol: How the FTP protocol Challenges Firewall Security*. URL: http://www.isaserver.org/articles-tutorials/articles/How_the_FTP_protocol_Challenges_Firewall_Security.html.

[25] *FTP4j Official Implementation Description*. URL: http://www.sauronsoftware.it/projects/ftp4j/.

[26] *FTPClient, Formal Implementation Description*. URL: https://commons.apache.org/proper/commons-net/javadocs/api-1.4.1/org/apache/commons/net/ftp/FTPClient.html.

[27] *Gibs W.W., Softwares Chronic Crisis, Scientific American, September, 1994*.

[28] Alain Girault, Bilung Lee, and Edward A. Lee. "Hierarchical finite state machines with multiple concurrency models." In: *IEEE Trans. on CAD of Integrated Circuits and Systems* 18.6 (1999), pp. 742–760. DOI: 10.1109/43.766725. URL: http://dx.doi.org/10.1109/43.766725.

[29] E. Mark Gold. "Complexity of Automaton Identification from Given Data." In: *Information and Control* 37.3 (1978), pp. 302–320. DOI: 10.1016/S0019-9958(78)90562-4. URL: http://dx.doi.org/10.1016/S0019-9958(78)90562-4.

[30] Serge Gorbunov and Arnold Rosenbloom. "AutoFuzz : Automated Network Protocol Fuzzing Framework." In: *IJCSNS International Journal of Computer Science and Network Security* 10.8 (2010), pp. 239–245. URL: https://cs.uwaterloo.ca/~sgorbuno/publications/autofuzz.pdf.

[31] D. Goswami and K. V. Krishna. "Formal languages and automata theory." In: November (2010), pp. 1–7. eprint: https://www.iitg.ernet.in/dgoswami/Flat-Notes.pdf.

[32] Nasa Goverment. *Mars Climate Orbiter team finds likely cause of loss*. URL: http://mars.nasa.gov/msp98/news/mco990930.html.

[33] B. Grubb. *Heartbleed Disclosure Timeline: Who Knew What and When*. April 2014. URL: http://www.smh.com.au/it-pro/security-it/heartbleed-disclosure-timeline-who-knew-what-and-when-20140414-zqurk.html.

[34] Siddharth Gujrathi. "Heartbleed Bug : AnOpenSSL Heartbeat Vulnerability." In: *International Journal of Computer Science and Engineering* 2.5 (2014), pp. 61–64. URL: http://www.ijcseonline.org/pub_paper/IJCSE-00277.pdf.

[35] Andreas Hagerer et al. "Model Generation by Moderated Regular Extrapolation." In: *Lecture Notes in Computer Science* (2002), pp. 80–95. DOI: 10.1007/3-540-45923-5{_}6.

[36] Guy Helmer et al. "A Software Fault Tree Approach to Requirements Analysis of an Intrusion Detection System." In: *Requirements Engineering* 7.4 (2002), pp. 207–220. ISSN: 09473602. DOI: 10.1007/s007660200016.

[37] Gerard J. Holzmann, Doron A. Peled, and Mihalis Yannakakis. "On nested depth first search." In: *The Spin Verification System, Proceedings of a DIMACS Workshop, New Brunswick, New Jersey, USA, August, 1996.* 1996, pp. 23–32.

[38] *How Machine Learning Could Result In Great Applications*. URL: https://datafloq.com/read/machine-learning-result-great-applications-busines/120.

[39] Jr. Howard F. Didsbury. *Future Vision: Ideas, Insights, and Strategies*.

[40] Malte Isberner, Falk Howar, and Bernhard Steffen. "The Open-Source LearnLib - A Framework for Active Automata Learning." In: *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*. 2015, pp. 487–495. DOI: 10.1007/978-3-319-21690-4_32. URL: http://dx.doi.org/10.1007/978-3-319-21690-4_32.

[41] *Issue has been reported to Apache Developers*. URL: https://issues.apache.org/jira/browse/NET-590?page=com.atlassian.jira.plugin.system.issuetabpanels:all-tabpanel.

[42] Jan Jürjens. "Model-based Security Testing Using UMLsec." In: *Electronic Notes in Theoretical Computer Science* 220 (2008), pp. 93–104. ISSN: 15710661. DOI: 10.1016/j.entcs.2008.11.008.

[43] Gonzalez Alberto Gross Hans-Gerhard Kanstren Teemu Piel Eric. "Observation-Based Modeling for Testing and Verifying Highly Dependable Systems A Practitioners Approach." In: *http://archiv.ub.uni-heidelberg.de/volltextserver/10095/* (), pp. 1–8.

[44] Yum K.Kwan and Gregory C. Chow. "Chow's method of optimal control: A numerical solution." In: *Journal of Economic Dynamics and Control* 21.4-5 (1997), pp. 739–752. ISSN: 01651889. DOI: 10.1016/S0165-1889(96)00005-X. URL: http://linkinghub.elsevier.com/retrieve/pii/S016518899600005X.

[45] *LearnLib: an API Tutorial*.

[46] *LearnLib Features*. URL: http://learnlib.de/features/.

[47] *LearnLib Studio*. URL: http://ls5-www.cs.tu-dortmund.de/projects/learnlib/index.php.

[48] *LearnLib Tool*. URL: http://learnlib.de/features/.

[49] *LearnLib Tool, a Tutorial*. URL: http://learnlib.de/wp-content/uploads/2013/08/learnlib.pdf.

[50] Alexander Maier. "Online passive learning of timed automata for cyber-physical production systems." In: *Proceedings - 2014 12th IEEE International Conference on Industrial Informatics, INDIN 2014* (2014), pp. 60–66. DOI: 10.1109/INDIN.2014.6945484.

[51] School of Electrical Engineering Mark Allman Shawn Ostermann and Computer Science Hans Kruse. "Data Transfer efficiency over satellite circuits using a multi-socket extension to the File Transfer Protocol (FTP)." In: *Advanced Communication Technology Satellite Results Conference* (). URL: http://ntrs.nasa.gov/search.jsp?R=19960052628.

[52] Aditya P Mathur. "Foundations of Software Testing." In: (2013).

[53] Karl Meinke and Fei Niu. "Learning-Based Testing for Reactive Systems Using Term Rewriting Technology." In: *Testing Software and Systems - 23rd IFIP WG 6.1 International Conference, ICTSS 2011, Paris, France, November 7-10, 2011. Proceedings*. 2011, pp. 97–114. DOI: 10.1007/978-3-642-24580-0_8. URL: http://dx.doi.org/10.1007/978-3-642-24580-0_8.

[54] Karl Meinke, Fei Niu, and Muddassar A. Sindhu. "Learning-Based Software Testing: A Tutorial." In: *Leveraging Applications of Formal Methods, Verification, and Validation - International Workshops, SARS 2011 and MLSC 2011, Held Under the Auspices of ISoLA 2011 in Vienna, Austria, October 17-18, 2011. Revised Selected Papers*. 2011, pp. 200–219. DOI: 10.1007/978-3-642-34781-8_16. URL: http://dx.doi.org/10.1007/978-3-642-34781-8_16.

[55] Karl Meinke and Muddassar A. Sindhu. "Incremental Learning-Based Testing for Reactive Systems." In: *Tests and Proofs - 5th International Conference, TAP 2011, Zurich, Switzerland, June 30 - July 1, 2011. Proceedings*. 2011, pp. 134–151. DOI: 10.1007/978-3-642-21768-5_11. URL: http://dx.doi.org/10.1007/978-3-642-21768-5_11.

[56] Maik Merten. "Active automata learning for real life applications." PhD thesis. Dortmund University of Technology, 2013. URL: http://hdl.handle.net/2003/29884.

[57] Maik Merten et al. "Automated Learning Setups in Automata Learning." In: *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change - 5th International Symposium,*

ISoLA 2012, Heraklion, Crete, Greece, October 15-18, 2012, Proceedings, Part I. 2012, pp. 591–607. DOI: 10.1007/978-3-642-34026-0_44. URL: http://dx.doi.org/10.1007/978-3-642-34026-0_44.

[58] Kumpati S. Narendra and M. a. L. Thathachar. "Learning Automata - A Survey." In: *IEEE Transactions on Systems, Man, and Cybernetics* SMC-4.4 (1974), pp. 323–334. ISSN: 0018-9472. DOI: 10.1109/TSMC.1974.5408453.

[59] Srinivas Nidhra and Jagruthi Dondeti. "Black Box and White Box Testing Techniques - A Literature Review." In: *International Journal of Embedded Systems and Applications* 2.2 (2012), pp. 29–50. ISSN: 18395171. DOI: 10.5121/ijesa.2012.2204.

[60] Ann Nowé, Katja Verbeeck, and Maarten Peeters. "Learning Automata as a Basis for Multi Agent Reinforcement Learning." In: *Proceedings of the 4th European Workshop on Multi-Agent Systems EUMAS'06, Lisbon, Portugal, December 14-15, 2006*. 2006. URL: http://ceur-ws.org/Vol-223/11.pdf.

[61] Ann Nowé, Katje Verbeeck, and Maarten Peeters. "Learning automata as a basis for multi agent reinforcement learning." In: *CEUR Workshop Proceedings* 223 (2006), pp. 71–85. ISSN: 16130073.

[62] *OpenSSL Wiki*. URL: https://wiki.openssl.org/index.php/SSL_and_TLS_Protocols.

[63] Corina S. Pasareanu et al. "Learning to divide and conquer: applying the L* algorithm to automate assume-guarantee reasoning." In: vol. 32. 3. 2008, pp. 175–205. DOI: 10.1007/s10703-008-0049-6. URL: http://dx.doi.org/10.1007/s10703-008-0049-6.

[64] Plato. *Plato Quotes*. URL: http://www.goodreads.com/quotes/359959-you-know-that-the-beginning-is-the-most-important-part.

[65] Harald Raffelt et al. "LearnLib: A framework for extrapolating behavioral models." In: *International Journal on Software Tools for Technology Transfer* 11.5 (2009), pp. 393–407. ISSN: 14332779. DOI: 10.1007/s10009-009-0111-8.

[66] *Request For Comments (RFC)*. URL: https://www.ietf.org/rfc.html (visited on 03/22/2015).

[67] Dana Ron. "Automata Learning and its Applications." In: (1995).

[68] Joeri de Ruiter and Erik Poll. "Protocol State Fuzzing of TLS Implementations." In: *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015*. 2015, pp. 193–206. URL: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/de-ruiter.

[69] Muddassar Azam Sindhu. *Algorithms and Tools for Learning-based Testing of Reactive Systems*. 2013. ISBN: 9789175016740.

[70] Bernhard Steffen, Falk Howar, and Maik Merten. "Introduction to Active Automata Learning from a Practical Perspective." In: *Formal Methods for Eternal Networked Software Systems - 11th International School on Formal Methods for the Design of Computer, Communication and Software Systems, SFM 2011, Bertinoro, Italy, June 13-18, 2011. Advanced Lectures*. 2011, pp. 256–296. DOI: 10.1007/978-3-642-21455-4_8. URL: http://dx.doi.org/10.1007/978-3-642-21455-4_8.

[71] Instructor Cliff Stein. "Hopcroft-Karp Bipartite Matching Algorithm and Hall Theorem Hopcroft-Karp Algorithm." In: (2012), pp. 1–4.

[72] *TCP Protocol, Formal Description from RFC 793*. URL: https://tools.ietf.org/html/rfc793.

[73] *The Heartbleed Bug*.

[74] *The jABC Interface*. URL: http://hope.scce.info/the-jabc4-interface/.

[75] *The Mattie J.T Stepanek Foundation*. URL: http://www.mattieonline.com.

[76] *TLS Protocol*. URL: https://www.sans.org/reading-room/whitepapers/protocols/ssl-tls-beginners-guide-1029.

[77] *Tomte, introducing the SUT Tool*. URL: http://tomte.cs.ru.nl/Sut-0-4/Description.

[78] *Tomte, Official description of the Tool*. URL: http://tomte.cs.ru.nl.

[79] *Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS) Heartbeat Extension*. URL: http://tools.ietf.org/html/rfc6520,urldate={October1985}.

[80] *Transport Layer Security (TLS), Formal Description from RFC 5246*. URL: https://tools.ietf.org/html/rfc5246#page-37.

[81] Jan Tretmans. "Model-Based Testing and Some Steps towards Test-Based Modelling." In: *Formal Methods for Eternal Networked Software Systems - 11th International School on Formal Methods for the Design of Computer, Communication and Software Systems, SFM 2011, Bertinoro, Italy, June 13-18, 2011. Advanced Lectures*. 2011, pp. 297–326. DOI: 10.1007/978-3-642-21455-4_9. URL: http://dx.doi.org/10.1007/978-3-642-21455-4_9.

[82] *Vsftpd Server, Formal Description*. URL: https://security.appspot.com/vsftpd.html.

[83] Stephan Windmüller et al. "Active continuous quality control." In: *CBSE'13, Proceedings of the 16th ACM SIGSOFT Symposium on Component Based Software Engineering, part of Comparch '13, Vancouver, BC, Canada, June 17-21, 2013*. 2013, pp. 111–120. DOI: 10.1145/2465449.2465469. URL: http://doi.acm.org/10.1145/2465449.2465469.

[84] *Wireshark, Official Description of the Tool*. URL: http://wireshark.org.

[85] Barbara Dosher Zhong-Lin Lu. *Visual Psychophysics: From Laboratory to Theory*.