MŰEGYETEM 1782

DIPLOMATERV FELADAT

Székely Gábor
szigorló mérnök informatikus hallgató részére

Protokoll állapotgépek visszafejtése

Számos alkalmazási környezetben használnak zárt protokollokat, melyek specifikációja nyilvánosan nem elérhető. Ugyanakkor biztonsági szempontból hasznos lenne ezen protokollok üzeneteinek és működésének értelmezése, mert ez lehetővé tenné a gyanús működés és a potenciálisan támadó jellegű üzenetek azonosítását, szűrését, és az ilyen események hatására riasztások generálását. Röviden, a hálózati forgalom folyamatos megfigyeléséhez és értelmezéséhez, támadások és anomáliák detektálásához általában szükséges a hálózatban használt protokollok üzeneteinek és működésének ismerete. Többen foglalkoztak már zárt protokollok üzenet típusainak és az egyes üzenet típusok formátumának visszafejtésével, ám kevesebb munka található az ismeretlen protokoll állapotgépének visszafejtésével kapcsolatban.

A hallgató feladata a protokoll állapotgép visszafejtéssel kapcsolatos irodalom áttekintése, olyan módszer azonosítása, mely alkalmas lehet ICS/SCADA környezetben használt protokollok állapotgépeinek visszafejtésére, az azonosított módszer vizsgálata és esetleges továbbfejlesztése, majd a továbbfejlesztett módszer prototípus szintű implementációjával egy olyan gyakorlatban használható rendszer kifejlesztése, mely jól támogatja a protokoll visszafejtéssel foglalkozó szakemberek munkáját. A feladat része továbbá a javasolt módszer értékelése, korlátainak azonosítása is. A feladat megoldása során feltételezhető, hogy az üzenet típusok és formátumok visszafejtése már megtörtént, azaz az üzenet típusok és formátumok valamilyen pontossággal már ismertek, és csak a protokoll állapotgép ismeretlen.

Tanszéki konzulens: Dr. Buttyán Levente, docens
Külső konzulens:

Budapest, 2020. március 10.

Dr. Imre Sándor
tanszékvezető, egyetemi tanár

Konzulensi vélemények:

Tanszéki konzulens: ☐ Beadható, ☐ Nem beadható, dátum: aláírás:

Külső konzulens: aláírás:

MŰEGYETEM 1782

Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Networked Systems and Services

Reverse engineering protocol state machines

MSc Thesis

Author
Gábor Székely

Supervisor
Dr. Levente Buttyán

December 10, 2020

Contents

HALLGATÓI NYILATKOZAT

Budapest, December 10, 2020

Gábor Székely
hallgató

Abstract

Over the past 50 years computers became increasingly connected with the use of various technologies, forming ever growing networks. In order computers to be able to communicate with each other, they need to use a common language, a protocol. With the growth of computer networks come various different protocols, many of which do not have open specifications. This hinders the development of compatible applications, safety and security verification and the development of network monitoring and protection software and policies.
When faced with one of the above tasks, one is left with the only option of reverse engineering the protocol, which is a quite challenging task as demonstrated by the SAMBA project that reverse engineered the SMB protocol requiring over 12 years to complete. To solve such hard problems, many automatized tools and methods that help with the process of reverse engineering protocols have been developed, creating the research area of Automatic Protocol Reverse Engineering (APRE). Most of the APRE tools aim to recover either the syntax of the messages of the protocol or protocol state machine. The protocol state machine describes how different messages in the protocol can follow each other.
In my thesis, I present an algorithm for automatically reverse engineering the state machine of protocols. At the core, I utilize a known algorithm, ${L_M}^+$, for inferring Mealy machines, by transforming the network messages into input and output letters that the Mealy machine and ${L_M}^+$ algorithm can understand. Then, to improve coverage, a random or guided search for previously unknown messages with different behaviour is added. An additional algorithm for simplifying the resulting Mealy machine is presented as well. The algorithm is implemented and tested on two real world protocols: Modbus and MQTT.

Kivonat

Az elmúlt 50 év során a számítógépek különböző technológiák segítségével egyre jobban összekötötté váltak, egyre növekvő hálózatokat alkotva. A számítógépek kommunikációját kommunikációs protokollok fejlesztésével tesszük lehetővé, melyek nagy hányadának nem nyíltan elérhető a specifikációja. Ez hátráltatja a kompatibilis alkalmazások fejlesztését, a biztonsági tesztelést, illetve a hálózatot monitorozó és védő szoftverek és házirendi szabályok létrehozását.

A fenti feladatok megoldásához nincs más választásunk, mint a protokollok visszafejtése, mely, ahogyan a SAMBA projekt is demonstrálta egy kifejezetten nehéz feladat. A SAMBA projekt keretében az SMB protokollt fejtették vissza mintegy 12 év munkája során. Annak érdekében, hogy felgyorsítsák és megkönnyítsék az ilyen feladatokat, különböző automatikus protokoll visszafejtési (APRE) módszereket fejlesztettek, megteremtve ezzel egy új kutatási területet. A legtöbb APRE eszköz vagy a protokoll üzeneteinek szintaxisát, vagy a protokoll állapotgépét próbálja automatikusan megadni. Egy protokoll állapotgépe azt írja le, hogy hogyan követhetik a különböző üzenetek a kommunikáció során.

A dolgozatomban egy olyan algoritmust dolgozok ki, mely képes automatikusan visszafejteni a kommunikációs protokollok állapotgépét. Az algoritmus magja az ${L_M}^+$ algoritmust használja fel, mely egy ismert algoritmus Mealy automaták automatikus tanulására. Az ${L_M}^+$ a hálózati üzenetek a Mealy automata és az ${L_M}^+$ algoritmus által érthető be- és kimeneti betűk közötti oda- és vissza-transzformálásával van felhasználva. Emellett, a lefedettség növelésének érdekében véletlenszerű, illetve vezérelt keresést végzek olyan üzenetekért, amelyek még nem felfedezett működést produkálnak. Az eredményül kapott Mealy automaták könnyebb értelmezhetősége érdekében egy utófeldolgozási algoritmust is definiálok. Az algoritmusokat implementáltam és két valódi protokollon, a Modbuson és MQTT-n teszteltem is.

Chapter 1

Introduction

A communication protocol defines the format and rules of information exchange between two entities. The entities could be various things, such as two programs running on the same computer, or they could be on different hosts, far away from each other, connected through a network. Other examples could be an Internet of Things (IoT) device communication with a sensor using a serial connection, or the IoT device then forwarding the measurement data to a central server through some wireless connection.

Almost all programs and computing devices that are used today use some kind of communication protocol, and it seems, that things are getting more and more connected with internet access getting cheaper and faster and the emergence of cheap IoT and smart devices. It directly follows, that any new application or device will have to support some of the existing protocols, in order to be able to cooperate with the existing infrastructure. In some cases the specifications for the protocols are open, or can be purchased, making it possible to develop the necessary program code for implementing the communication routines. For some widely used protocols there are even free and open source implementations that can be reused, saving many hours of work for the developer. In a lot of other cases though, unfortunately, there is little to no information available to the public, making the development of compatible software extremely difficult. There can be many reasons for this: for example the documentation of the protocol may simply not exist, or it could be out of date, due to scarce resources. In other cases the creators do not share the existing specification on purpose, for example to maintain a competitive advantage or to make interoperability harder. As a real world example, the Microsoft Server Message Block (SMB) which is a network protocol mainly used in Windows networks for various purposes, was a closed source, proprietary protocol for a long time, before finally opening up the specification. The SAMBA project is an open source implementation of this protocol and illustrates the difficulty of reverse engineering protocols, as it took over 12 years to complete[1].

However, there are other motivations as well for reverse engineering communication protocols. The huge importance of networks also necessitates serious protections for them. It is not possible to make sure that a protocol is safe, without understanding how it works. Thus, reverse engineering the protocol allows researchers to verify that there are no vulnerabilities in it. It also allows the development of network traffic monitoring appliances, such as Application Level Firewalls, or Intrusion Detection Systems (IDS). These systems monitor network traffic and parse the various protocols to extract high level information, that can help prevent or detect attacks on the network. Knowledge of the protocol specification is necessary to construct the parsers that make this possible. Furthermore, malicious botnets also need protocols to communicate with members of their network and the command and control server. These protocols are usually custom, thus making it harder for security researchers and system administrators to detect and stop attacks launched by these botnets. Reverse engineering these protocols allows for better insight into the activities of such botnets.

As mentioned previously, in the SAMBA project, the SMB protocol was reverse engineered mostly manually, although the previously linked page mentions some automatized processes as well. Some of the methods used were the following: publicly available resources – although often incomplete or inaccurate – and deducing the meaning of messages by monitoring network traffic while executing some known function. More automatized methods include trying to fuzz some value inside a message and deducing information from the error messages returned from every version; and comparing a custom implementations response to the response of original implementation. As it became evident, that protocol reverse engineering is an important problem, several approaches were developed, to automatize the process. Mostly reffered to as Automatic Protocol Reverse Engineering or APRE for short, these solutions usually aim at determining the format/syntax of protocol messages and recovering the state machine of the protocol. The syntax describes how valid messages are composed, and the possible values of the fields, possibly even their meaning. In turn the state machine of the protocol describes how different messages can follow each other in a session of the protocol. In most cases an APRE method only targets one of these goals, so one has to either combine multiple methods, or complement them with manual reverse engineering where necessary.

Another application, that is actually specific to APRE is verifying and comparing

[1]`https://www.samba.org/ftp/tridge/misc/french_cafe.txt` (last checked 2020.11.29)

implementations of protocols. If the original specification is available, and we apply an APRE technique, we can then compare the output of the tool on the implementations to the original specification. Similarly, even if there is no open specification, we can compare two implementations, by evaluating them both with the same APRE tool, and then looking for differences in the output. This can be helpful for example, to determine whether the two implementations can cooperate.

Further dividing APRE methods, they can be static, dynamic or interactive. A static method only uses recorded network traffic, a dynamic method requires a binary and usually analyzes it while it is running, and finally an interactive approach has black-box access to an implementation and can interact with it to gather information. The approach described in this work attempts to recover the state machine of the protocol with an interactive approach. As such, it needs black-box access to an implementation of the protocol, and in addition the message formats are also considered to be known beforehand, at least to an extent that makes it possible to determine the types of messages. A message type is a groups of messages that generally have the same function and only differ in parameters, counters, etc. The solution presented builds on a well researched algorithm of automata theory, to produce a Mealy machine that describes the protocol state machine. Assuming a server-client protocol, the input letters would correspond to the message type of queries from the client to the server and the corresponding output letter would be the message type of the response of the server. Additional measures are taken to make sure all versions of each message type that produce unique outputs are discovered by extending the original algorithm and running it multiple times with different sets of messages. Some additional methods are explored that aim to make human comprehension of complicated Mealy machines easier.

A publication [12] has also been made from the research presented in this thesis, while an extended version is currently under review.
The rest of the work is structured in the following way: in chapter 2, I explore existing research in the area of APRE, in chapter 3, I detail the necessary automata theory definitions and algorithms that are necessary for understanding the methods used, then describe how I adapted and extended the known algorithms. In chapter 4, I present my implementation of the algorithms that were outlined in the previous chapter, and in chapter 5, I present the results of the evaluation of the implementation. Finally in chapter 6, I summarize my results.

Chapter 2

Related work

Reverse engineering is a complicated, but essential process in most security related tasks. To aid experts in this task, many automatized and assisting tools have been developed. [13] aims to answer how proficient reverse engineers work, in order help guide development of tools to best complement the work of experts. Although the interviewees were practicing binary reverse engineering and not protocol reverse engineering the main results may be useful for APRE tool design as well. [13] is an observational study with 16 participants who were asked to share the process they used recently to reverse engineer a binary, by showing and reenacting how they completed their task. The main conclusion is that the reverse engineering process can be modeled with three main processes: overview, subcomponent-scanning, and focused experimentation. Each phase feeds information into the next one, and the last two steps may be repeated multiple times, with each step refining the understanding of the program that is being reversed. In overview, the expert gathers information about the program, such as how its user interface works, what features it has, what strings does the binary contain, etc. The information gathered here allows them to form hyptheses and questions about the program that they will need to test. In binary reverse engineering usually the whole program does not need to be understood completely, rather some specific questions need to be answered, such as: *Is this a malware?* or *How does this malware avoid detection?* The information gathered in overview allows the reverse engineer to select areas for subcomponent-scanning, where only parts of the program are analyzed in greater detail. Through subcomponent-scanning the initial hypotheses and questions are refined. These question and hypotheses are then tested through focused experimentation.

In the recent years there has been extensive research into APRE. [10] divides approaches first by the targeted part of the protocol: they can either target the syntax of the protocol, or its state machine. Syntax APRE can be further divided into network trace based and dynamic techniques. Tools that belong to this category only receive recorded network traffic as input and use variants and combinations of statistical approaches, clustering and string alignment techniques to recover the message formats of the target protocol. Some approaches that belong here are the following: Discoverer [7], Biprominer [15], AutoReEngine [14], ReverX [2] and GrAMeFFSI [9]. Dynamic techniques also utilize information about the execution of the program that implements the target protocol. Memory is monitored to see how bytes received are transformed and used. Tools in this category include AutoFormat [8], Dispatcher [3] and ReFormat [17].

The other main category, protocol state machine reversing can be divided into categories similarly. First, static approaches based on captured network traffic: ReverX [2] handles both syntax and protocol state machine reversing. This method assumes a client-server architecture. After determining the syntax and assigning message types based on syntax, it analyzes sessions of communication and creates partial FSMs from them. These FSMs are then merged and simplified to produce the final output. Another static method, Veritas [16] identifies keyword by looking for substrings that reappear with a high probability. A modified Kolmogorov-Smirnov test is used to filter out keywords that do not influence the protocol state machine. Based on these keywords message types are assigned with a *Partitioning Around Medoids* (PAM) algorithm and a probabilistic finite state machine is created by calculating the probability of one message type transitioning to another for all message types.

The dynamic category is also present, an example from this category is Prospex [6]. Prospex clusters messages based on their format, execution similarity (system calls, library calls and memory locations) and server impact (for example how long does the server take to calculate the response). A PAM algorithm is used to place these messages into groups and an FSM is constructed.

Finally, an an additional category which allows black-box access to the implementation is also present for protocol state machine reversing. [18] and [5] are such approaches, in fact [5] uses the same algorithm from automata theory as the approach presented in this thesis, to reverse engineer the protocol of the MegaD botnet and even find vulnerabilities in its design that could disable the network. [5] proposes optimizations, to speed up the inference process: caching, parallelization and predicting the responses to queries. Another example of Angluin style Mealy machine learning is [4], using it to reverse engineer the state machine of a smart card reader. In this paper, a Lego robot is used to input PIN codes, and push *OK* or *Cancel* buttons on the physical interface of the card reader, enabling automated querying. Using this algorithm has the advantage of guaranteeing to produce the correct and minimal automata. The main improvement in my approach is that instead of only using the core algorithm a single time, while my approach completes several executions of it to attempt to create more subgroups of messages and increase coverage.

Chapter 3

Design

3.1 Automata theory overview

The solution I present in this thesis relies heavily on language and automata theory. In this section I introduce the concepts, definitions, and algorithms that are necessary to understand the ideas and solutions presented here. The following definitions and theorems are part of the *Languages and Automata* curriculum, and as such, the official lecture notes[1] were used as reference.

3.1.1 Basic definitions

Definition 3.1 (Alphabet)**.** *An arbitrary, non-empty set is an alphabet, usually denoted by* Σ*.*

Definition 3.2 (Letter)**.** *The elements of* Σ *are the letters or characters.*

As an example, the set containing a, b, and c is an alphabet, and the individual letter a is a letter (of course so are b and c).

Definition 3.3 (Word)**.** *A word is a finite sequence of letters.* Σ^* *denotes all the possible words that can be composed from the* Σ *alphabet. Note that a zero length word is also valid, usually denoted by* ε*.*

Continuing the previous example, any combination of the three letters can be a word, for example, all of the following are words: a, abc or $aabac$.

[1] `http://www.cs.bme.hu/~friedl/nyau/jegyzet-13.pdf` and `http://www.cs.bme.hu/~Efriedl/nyau/kimenetes-uj.pdf` (last visited: 2020.11.26)

Definition 3.4 (Formal language)**.** *A formal language L over the alphabet Σ, is a – finite or infinite – set of words that can be composed from the letters in Σ.*

A language over the previously defined alphabet, could be the set composed of the three words in the previous example. An example for an infinite alphabet could be the set of all the words in which the letters follow each other in alphabetical order (denoted as L_{alph} from here on). This language would contain *abc* and *aac* and many more, but *ca* would not be part of this language. However it is rather inconvenient to always list all the words that belong to a language, and even impossible, if the language has infinite words. The following tools provide a way to formally define both finite and infinite languages.

Definition 3.5 (Formal grammar)**.** *A formal grammar $G = (V, \Sigma, S, P)$ is a system, where*

- *V is a finite, non-empty set, containing the variables,*
- *Σ is an alphabet, $V \cap \Sigma = \emptyset$,*
- *$S \in V$ is the starting variable,*
- *P is a finite set that contains the production rules. The elements of P are of the form $\alpha \to \beta$, where α and β are sequences composed of the elements of V and Σ, and α contains at least one variable.*

Definition 3.6 (Derivation)**.** *A derivation in grammar G is a*

$$\gamma_0 \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \cdots \Rightarrow \gamma_n$$

finite sequence, where $\gamma_0 = S$, $\gamma_i \in (V \cup \Sigma)^*$ and every γ_{i+1} can be produced from γ_i with a production rule for every i. This means, that there is a $\delta_1, \delta_2, \alpha, \beta \in (V \cup \Sigma)^*$, $\alpha \to \beta \in P$, where γ_i and γ_{i+1} can be divided into $\gamma_i = \delta_1 \alpha \delta_2$ and $\gamma_i = \delta_1 \beta \delta_2$.

Definition 3.7 ($L(G)$)**.** *$L(G)$ is the language generated by G. It consists of $w \in \Sigma^*$ words that can be derived in G starting from S:*

$$S \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \cdots \Rightarrow w$$

Now we can clearly define L_{alph} from the previous example. Let us take the grammar $G_{alph} = (\{S, B, C\}, \{a, b, c\}, S, P)$, where $P = \{S \to aS,\ S \to B,\ B \to bB,\ B \to$

$C,\ C \to \varepsilon,\ C \to cC\}$. This is often written in a shorter and easier to read form:

$$\begin{aligned} S &\to aS \mid B \\ B &\to bB \mid C \\ C &\to \varepsilon \mid cC \end{aligned}$$

At any point in the derivation, there is at most one variable, and each variable can only come after the previous one, we cannot go back after changing to the next variable. This guarantees the alphabetical order. The ε production rule allows us to produce the empty word. Thus $L(G_{alph}) = L_{alph}$ As an example, let us take a look at how the word *aac* can be generated:

$$S \to aS \to aaS \to aaB \to aaC \to aacC \to aac$$

3.1.2 Chomsky hierarchy

By imposing additional restrictions on the production rules of a grammar, they can be further categorised. Noam Chomsky created four classes of grammars, the higher the class, the more the restrictions. More restrictions result in being able to describe only simpler languages, but at the same time, the corresponding automaton that can detect the language is also simpler.

Definition 3.8 (3. class)**.** *In a class 3 grammar, all production rules are of the form* $A \to aB$ *or* $A \to a$*, where* $A, B \in V$ *and* $a \in \Sigma$ *are arbitrary. In addition the* $S \to \varepsilon$ *rule is allowed for the starting variable, if* S *is not present on the right side of any rule.*

Definition 3.9 (2. class)**.** *In a class 2 grammar, all production rules are of the form* $A \to \alpha$*, where* $A \in V$ *and* $\alpha \in (V \cup \Sigma)^*$ *are arbitrary as long as* α *is at least one in length. In addition the* $S \to \varepsilon$ *rule is allowed for the starting variable, if* S *is not present on the right side of any rule.*

Definition 3.10 (1. class)**.** *In a class 1 grammar, all production rules are of the form* $\beta A \gamma \to \beta \alpha \gamma$*, where* $A \in V$ *and* $\alpha, \beta, \gamma \in (V \cup \Sigma)^*$ *are arbitrary as long as* α *is at least one in length. In addition the* $S \to \varepsilon$ *rule is allowed for the starting variable, if* S *is not present on the right side of any rule.*

Definition 3.11 (0. class)**.** *In a class 0 grammar the only restriction is that on the left side of the rules have to contain at least one variable.*

A class 3 grammar is usually called regular grammar, L_{alph} can also be described with a regular grammar:

$$\begin{aligned}
S &\rightarrow \varepsilon \mid a \mid b \mid c \mid aA \mid aB \mid aC \mid bB \mid bC \mid cC \\
A &\rightarrow a \mid aA \mid aB \mid aC \\
B &\rightarrow b \mid bB \mid bC \\
C &\rightarrow c \mid cC
\end{aligned}$$

A class 2 grammar is usually called a context free (CF) grammar, an example that cannot be described with a regular grammar but is possible with CF grammar is the language that has the same amount of each letter and the letters in the words are in alphabetical order, over the alphabet $\{a, b\}$ (this can be denoted as $a^n b^n$):

$$\begin{aligned}
S &\rightarrow \varepsilon \mid ab \mid aPb \\
P &\rightarrow ab \mid aPb
\end{aligned}$$

A class 1 grammar is usually called a context sensitive (CS) grammar, an example that cannot be described with a CF grammar but is possible with CS grammar is the same language as before, but over the alphabet $\{a, b, c\}$ (this can be denoted as $a^n b^n c^n$):

$$\begin{aligned}
S &\rightarrow \varepsilon \mid abc \mid aPbc \\
P &\rightarrow abC \mid aPbC \\
Cb &\rightarrow bC \\
Cc &\rightarrow cc
\end{aligned}$$

3.1.3 Finite State Machines

Definition 3.12 (Finite State Machine). *A Finite State Machine (FSM) is an $M = (Q, \Sigma, \delta, q_0, F)$ tuple, where:*

- *Q is a finite, non-empty set, it contains the states of the automaton,*
- *Σ is a finite, non-empty set, the alphabet of the automaton,*
- *$\delta : Q \times \Sigma \rightarrow Q$ is the state transition function of the automaton,*

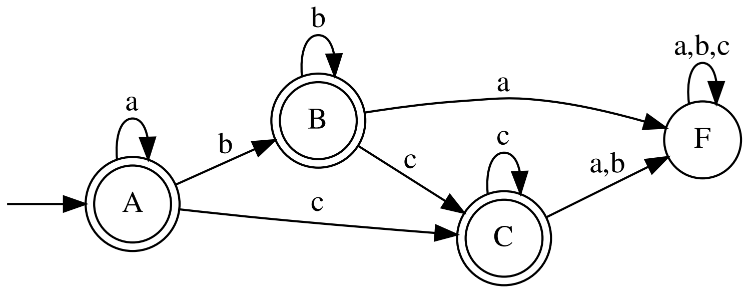

Figure 3.1. *M_{alph} in graph representation*

- *$q_0 \in Q$ is the starting state,*
- *$F \subseteq Q$ is the set of accepting states.*

An $r_0, r_1, \ldots, r_n (r_i \in Q)$ state sequence is the calculation executed on the word $w = a_1 a_2 \ldots a_n \in \Sigma^*$, if $r_0 = q_0$ and $r_i = \delta(r_{i-1}, a_i)$, for every $i = 1 \ldots n$. M accepts w if $r_n \in F$. $L(M)$, the language accepted by M, is the set of words accepted by M.

Definition 3.13 (Regular Languages)**.** *$L \subseteq \Sigma^*$ is a regular language if there exists M FSM, for which $L(M) = L$.*

Theorem 3.1. *There exists a class 3 grammar for L if, and only if L is regular.*

The proof is based around two algorithms that can transform between FSM and grammar, thus proving the equivalence. For example, using this algorithm, we can transform the example language L_{alph} that was used in subsection 3.1.2: $M_{alph} = (\{A, B, C, F\}, \{a, b, c\}, \delta, A, \{A, B, C\})$, where δ is: $\delta(A, a) = A,\ \delta(A, b) = B,\ \delta(A, c) = C,\ \delta(B, a) = F,\ \delta(B, b) = B,\ \delta(B, c) = C,\ \delta(C, a) = F,\ \delta(C, b) = F,\ \delta(C, c) = C,\ \delta(F, a) = F,\ \delta(F, b) = F,\ \delta(F, c) = F$.
FSMs can be drawn as graphs, which form is often used, as it is easier to understand if the machine is not too complicated. This form can be seen in Figure 3.1.

An important consequence of this theorem is that there are certain constructs that FSMs just cannot describe. Mealy machines, the machines my solution is based on are very similar to FSMs, thus this restriction in expression carries on to the results. For example, it is impossible to model $a^n b^n$ with them, so if there is no upper limit to the counting, a Mealy machine will not be enough. However the simplicity of regular languages makes it possible to infer them very well, and for many protocols they are good enough.

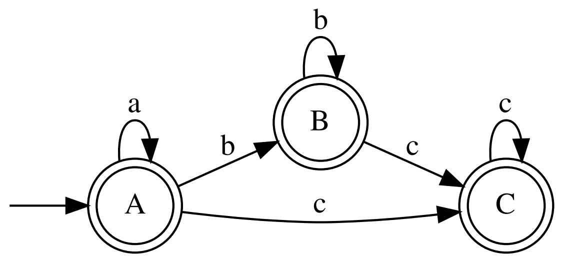

Figure 3.2. *M_{inc} in graph representation*

Definition 3.14 (Incomplete FSM)**.** *An Incomplete FSM is an FSM in which δ is not defined for all combination of states and input letters. If an undefined state transition would be taken, the word is rejected.*

Using an incomplete FSM can be useful to present an FSM in an easier to comprehend form. For example, an incomplete FSM for detecting L_{alph} can be seen in Figure 3.2.

Fully defined and incomplete FSMs are equivalent, because an incomplete FSM can always be converted into a complete FSM, such that the two FSMs detect the same language. Algorithm 3.1 accomplishes this task.

Algorithm 3.1 (Fully defining an FSM)**.** *First, we add a new state q_{FAIL} to Q, such that $q_{FAIL} \notin F$. Then, for every $q \in Q$, $a \in \Sigma$, where $\delta(q, a)$ is not defined (including the new state), we define it as $\delta(q, a) = q_{FAIL}$.*

In some cases, it is easier to define an FSM, if the state transitions can be undeterministic, i.e., it is possible to transition into more than one new state from a certain state for a certain input letter. FSMs where this is allowed are called undeterministic FSMs. Same as with incomplete FSMs, deterministic and undeterministic FSMs are equivalent, if an M undeterministic machine exists such that $L(M) = L$, then there is an M' deterministic machine such that $L(M') = L$. Formally, undeterministic FSMs work as defined below:

Definition 3.15 (Undeterministic FSM)**.** *The definition is identical to Definition 3.12, except for δ, which is now defined as $\delta(q, a) \subseteq Q$, where $q \in Q$ and $a \in \Sigma \cup \{\varepsilon\}$.*

Definition 3.16 (Undeterministic FSM calculation)**.** *An $r_0, r_1, \ldots, r_m (r_i \in Q)$ state sequence is the calculation for the $w = a_1, a_2, \ldots, a_n$ word, if*

1. $r_0 = q_0$*,*

2. at any point, after reading $a_0, a_1, \ldots, a_j$ *word, and being in state* r_i*, then* $r_i \in \delta(r_{i-1}, \varepsilon) \cup \delta(r_{i-1}, a_i)$ *(the automaton either gets to a new state by reading a letter and taking a transition with that letter as the input, or does not read anything and takes a transition with* ε*),*

3. the whole word has been read, when the automaton is in the state r_m *(there may have been* ε *transitions before* r_m *and after reading the whole word as well).*

The undeterministic FSM accepts w*, if there exists a calculation for it, in which* $r_m \in F$*. Otherwise it rejects it.*

There is an algorithm for determinizing an undeterministic FSM, that creates an equivalent deterministic FSM, by following all possible calculations. Basically, for every case, where the new state can be more than one state, a new combined state is created, which represents the possibility of being in either of the states. Then, outgoing transitions for the new state are created by combining all the transitions of the original states. All composite states that contain accepting states (states that are in F) will be accepting as well. The main downside to determinizing a machine is the increased number of states, which can be exponential.

The determinized FSM may contain more states than necessary, i.e., there might be an FSM that has fewer states than the one generated by the algorithm, but they detect the same language. Fortunately there is an algorithm for finding such FSMs.

Definition 3.17 (Minimal FSM)**.** *The minimal FSM for the* L *language is an* M *fully defined, deterministic FSM, such that* $L(M) = L$*, and there is no* M' *fully defined, deterministic FSM, such that* $L(M') = L$ *and the number of states in* M' *is smaller than in* M*.*

Theorem 3.2. *If there is an FSM for a language, there is a minimal FSM as well for it, and it is unequivocal.*

The proof for the theorem contains the algorithm for generating the minimal FSM from an FSM.

3.1.4 Automata with output

Mealy and Moore machines are a lot like FSMs, however instead of either accepting or rejecting words, they produce output. Many algorithms that work with FSMs

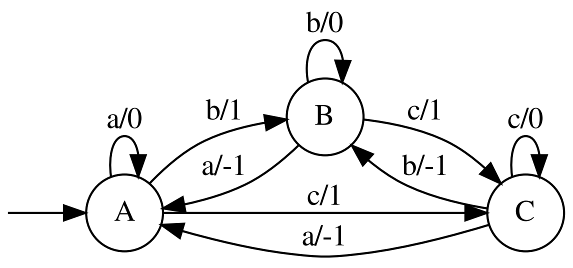

Figure 3.3. *M_{alph} in graph representation*

can be used on Mealy and Moore machines with minimal adjustments. Another similarity is that the two types are interchangeable, so it is possible to convert a Mealy-machine to a Moore machine and vice versa, all the while not changing their functionality. Because of this, since Mealy machines are better for representing protocol state machines, I only detail them.

Definition 3.18 (Mealy machine)**.** *A Mealy machine is a $M = (Q, \Sigma, \Delta, \delta, \lambda, q_0)$, where:*

- *Q is a finite, non-empty set, containing the states of the machine,*
- *Σ is a finite, non-empty set, the input alphabet of the machine,*
- *Δ is a finite, non-empty set, the output alphabet of the machine,*
- *$\delta : Q \times \Sigma \to Q$, is the state transition function,*
- *$\lambda : Q \times \Sigma \to \Delta$, is the output function,*
- *q_0 is the starting state.*

The Mealy machine works very similarly to a simple FSM, except for every state transition an output letter is produced, so by the end an output the same length as the input is generated. Mealy machines are usually displayed in graph form as well, with the input and output letters separated by a / character on the edges. An example can be seen in Figure 3.3, a Mealy machine M_{alph} that has $\Sigma = \{a, b, c\}$ for input alphabet and $\Delta = \{-1, 0, 1\}$ as output alphabet, and produces 0 output, if the input letter is the same as the previous letter, -1 if it is alphabetically earlier then the previous and 1 if larger. For the first input letter it acts as if an a was read before it.

3.1.5 Inferring automata

Apart from categorizing and translating words FSMs and Mealy machines are very good for modeling the behaviour of black-box systems as well. There has been extensive research into inferring or learning automata based on their behaviour or the words they accept/produce. In the setting first proposed by Angluin [1] the *learner* infers an FSM with the help of a *teacher* through queries and counterexamples. The teacher knows the FSM, and can answer two types of questions for the learner. First, the learner may ask *membership queries* from the teacher: the teacher will answer whether the FSM would accept the queried word. Second, the learner can present *conjectures*, which are FSMs that the learner thinks are the correct solution. The teacher either confirms the conjecture, or responds with a counterexample.

Angluin presents an efficient algorithm L^* for handling the problem, that outputs the minimal FSM that detects the same language as the one the teacher is trying to teach. L^* works by systematically querying the teacher for words and building a table from the results, which will directly generate the FSM. Each cell in the table represents whether the FSM would accept the word we get by concatenating the letters in the row and column header. The table is expanded with rows and columns to meet certain criteria (and the new cells are always filled based on membership queries), and when it does, a conjecture is produced from it. If the teacher responds with a counterexample, the table is expanded based on it and new membership queries until a new conjecture can be made. The algorithm stops when the teacher accepts the conjecture. For a complete description of the algorithm please refer to the original paper.

This version of the algorithm requires the teacher to know the FSM that the learner is trying to learn in order to be able to answer conjecture queries. However, in many applications this is not realistic. [1] proposes a solution for this: the teacher can check conjectures against a black-box FSM, that answers membership queries by generating a sufficiently large number of membership queries and comparing the output of the conjecture and the real FSM. It presents formula for determining the number of necessary queries given a desired confidence and accuracy level. This way the conjecture query of the teacher can be constructed using membership queries.

This learner-teacher model and Angluin's algorithm has since been thoroughly researched resulting in various improvements and adaptation to other types of automata, including Mealy machines. The algorithm ${L_M}^+$ that is applied in this thesis was published in [11], which includes some algorithmic improvements that reduce the number of membership queries necessary and an adoption for Mealy machines as well. The main difference in adapting the algorithm for Mealy machines is that in this case we use *output queries* that return the output of the Mealy machine for a certain word instead of *membership queries*; and the cells of the table contain the output of the FSM instead of a binary rejected/accepted information.

3.2 Protocol reverse engineering

3.2.1 Problem statement

A protocol specification can be divided into defining the protocols message formats and the protocol state machine. The message format defines what byte sequences are valid messages and possibly the meaning of the fields in a message. The protocol state machine defines how an interaction using the protocol can go: which messages come after which; what are valid and invalid sequences of messages and possibly what was done in the session. In my thesis, the general goal was recovering the protocol state machine, which I modeled with Mealy machines, where Σ is the set of possible query messages and Δ is the set of possible responses. The developed method generally assumes a client-server setup, where the client makes queries and the server responds to them, however cases where the server does not respond to certain queries or responds with multiple messages are also handled. Unprompted messages from the server are more problematic, but can also be handled.

There are different approaches to APRE of protocol state machines, such as static APRE, where only captured traffic is used, dynamic APRE, where various debug and control flow information is observed in addition while the implementation of the protocol is running, and interactive APRE, where various adaptively chosen messages are sent to host that is running an implementation of the protocol to deduce further information. Since my solution employs the ${L_M}^+$ algorithm at its core, it needs to be interactive. Two more capabilities are necessary because of the use of ${L_M}^+$ it is required to be able to reset the state of the server that is being queried and the environment needs to be deterministic. This is not to be confused with deterministic and undeterministic state machines, which are equivalent: it only means, that if a certain message sequence is sent to a freshly started server, it will always respond with with the same messages. Message formats are also necessary to be known beforehand at least to an extent, that allows categorization of messages. And lastly, although not strictly necessary, access to recorded network traffic can be used to improve the efficiency of the proposed algorithm.

The output of the algorithm is a Mealy machine, that can be used in an IDS to detect invalid flows of a protocol, or aid human comprehension and reverse engineering of the protocol, for which some post processing methods are also proposed, in order to produce cleaner and easier to understand Mealy machines.

3.2.2 Adapting the ${L_M}^+$ algorithm

The aim is to create a Mealy machine that represents how the protocol we are trying to reverse engineer works. It follows that input alphabet and the output should consist of the list of possible messages. Let us denote the set of possible messages in a protocol with $S_{messages}$. However simply filling the alphabets with all of the possible messages would be impractical: if a message contains even a single unrestricted integer field, for example to hold a measurement value returned by the server or to specify a request parameter sent to the server, such as setting some value, it would mean that $|\Sigma| >= 2^{32}$ and $|\Delta| >= 2^{32}$. This would result in ugly, unmanageable Mealy machines, not to mention very long running times of ${L_M}^+$ execution. To bring the sizes of alphabets to a manageable size, I define *message type* as follows.

Definition 3.19 (Message type)**.** *A message type is a set of messages in a protocol, that serves the same purpose and the individual messages only differ in their parameters. Let S_{mtype} denote the set of all message types in a protocol.*

Now the alphabets can be a lot smaller: $\Sigma \subseteq S_{mtype}$ and $\Delta \subseteq S_{mtype}$. It is necessary to know the syntax of the messages to an extent that allows us to create these message types. However, the better our understanding of the message syntax the better the precision of the result will be.

Three more *helper functions* are necessary to adapt the ${L_M}^+$ algorithm for learning protocols. The first is the message classifier: $f_{class} : S_{messages} \to S_{mtype}$. It is necessary in order to be able to translate the response sent by the server after making an *output query.* The second is the generator function: $f_{gen} : S_{mtype} \to S_{messages}$. f_{gen} may be a random function, meaning that calling it with the same message type can result in different messages being generated. Note that if recorded network traffic is available, it is possible to construct f_{gen} using f_{class} by classifying all the messages in the recording and randomly selecting one with the correct message type. The third is the updater function: $f_{upd} : m_0, m_1, \ldots, m_i \to m'_i$, it takes all the messages that have been sent in the current session and the current message that is about to be sent and updates it so counters and challenges are correctly matched.

With all the above functions given the ${L_M}^+$ algorithm can now be applied by creating a teacher that is able to answer a generic learner's queries. The learner needs to know Σ, so that is determined first, for example if a network trace is available by classifying the messages in it with f_{class} and initializing Σ to the set of message types that appeared as queries. Then we call f_{gen} for every element of Σ and store the results, which we will use inside the teacher to translate between message types and actual messages. This is necessary, because if we did not save the results, a random generator function may cause problems if it generated messages that may have different parameters resulting in different responses throughout the run of the algorithm. For example an integer parameter set below a certain value may result in an OK response, but if set above it, it could generate an error. Every time an output query is asked by the learner, the teacher resets the system under test (SUT), translates between the message types and the real messages using the saved values, uses f_{upd} to update the messages, sends them to the SUT, classifies the responses with f_{class} and returns the message types to the learner. The conjecture query can be constructed as described in subsection 3.1.5.

3.2.3 Multiple rounds extension

The results that the base adaptation of the ${L_M}^+$ algorithm as described in subsection 3.2.2 (from here on referred to as *base algorithm*) produces are heavily dependent on the quality of the message type categorization. For example, if there is a mistake in the categorization and two different message types are handled as one and the same, then only the one that was generated with f_{gen} at the start will appear in the output. But even if the message types are created perfectly, the output may still be incomplete because of the fixed messages: the server may respond differently to messages from the same message type if their parameters differ. For example, if a read command has a valid address the response will have the value, otherwise it can be an error message. In order to capture behaviour like that, the base algorithm needs to be extended further. First of all, we need to define what we want to detect:

Definition 3.20 (Message subtype)**.** *A message subtype is a set of messages that belong to the same message type and produce the same responses from the server as each other. Let $S_{msubtype}$ denote the set of every message subtype of a protocol.*

So in the previous example the read command message type would be divided into two subtypes: the ones with valid target addresses and the ones with invalid target addresses.

Since message subtyes can only be distinguished based on their behaviour, the base algorithm is executed with multiple messages from every message type. The aim is to distinguish subtypes by analyzing the output of the base algorithm. At the start of each run, f_{gen} is called a number of times, to expand Σ. For this to work, f_{gen} needs to be a random function, otherwise only the same message would be generated for each message type over and over again. At the start of each base algorithm run, it is assumed, that each message in Σ belongs to a different subtype, so in this version of the algorithm the input alphabet will be $\Sigma \subseteq S_{msubtype}$, while the output alphabet will remain $\Delta \subseteq S_{mtype}$. The base algorithm is run without any major changes, and the resulting Mealy machine is *deduplicated*, which I define as follows:

Definition 3.21 (Protocol state machine deduplication)**.** *When a Mealy machine is deduplicated, messages in Σ that belong to the same message type and produce the same behaviour are removed one by one, until no pair with the same message type and behaviour remains, meaning that each message subtype will only be represented by one message in Σ after performing deduplication.* Same behaviour *means, that the two input letters (a and b) are interchangeable: $\delta(q, a) = \delta(q, b)$ and $\lambda(q, a) = \lambda(q, b)$ for every $q \in Q$.*

Deduplication can be seen as a way to merge messages in Σ that belong to the same subtype into a single input letter. After deduplication, the assumption made at the beginning holds true. These steps – new message generation, base algorithm and deduplication – can be repeated as many times as necessary. Different types of criteria for stopping can be made, for example if there is no new subtypes found in a number of consecutive runs (the size of Σ after deduplication does not increase), the algorithm may stop. The number of new messages generated at the start of each round is also an adjustable parameter. The greater the number, the more messages are tested at the same time, however the greater $|\Sigma|$ is the more queries the ${L_M}^+$ will take to finish.

Using the base algorithm over and over again the subtypes can be gradually discovered by random search, without having to grow the size Σ to a too large value. It is important to note, that even though subtypes may be discovered, no function is created for classifying messages into subtypes. This is not a very big disadvantage in many cases, since the list of messages that belong to certain subtypes is often not specific to the protocol, rather it depends on the configuration of the server: the valid read addresses obviously depend on server settings, or a sign-in message can be divided into subtypes based on whether the credentials are correct or not. It is not a goal to map out valid address ranges or correct credentials, however this technique makes the different possible responses to message types such as these visible without the environment appearing to be undeterministic.

3.2.4 Improving message generation/selection

Choosing the number of new messages added at each round is an important decision: ideally, it should be kept as low as possible, but deciding on a too low number has its downsides as well. The first problem arises from the random nature of f_{gen}. If the distribution of the message subtype of the generated messages is uniform, i.e., the probability of the new message being a certain subtype is the same for all subtypes, then there is no problem. However if this is not the case, the algorithm may take many rounds to find the subtypes that have lower probability of being generated, or in a worst case scenario not find some subtypes at all before reaching the stopping criteria. If no care is taken to ensure a uniform distribution, f_{gen} is likely to have non-uniform distribution with regards to message subtypes. For example, the f_{gen} outlined in section 3.2.2 can be like this as well: assuming the previously outlined *read value* message and a well functioning system as an environment for the network trace, there would be a lot more messages with valid addresses, so f_{gen} would be more likely to return messages from the subtype with correct addresses. Using an f_{gen} function that just randomly fills out parameter fields would result in similarly unbalanced results if the number of valid and invalid addresses are not the same, not to mention the overwhelmingly more messages with valid credentials in the other example case.

The other problem with only adding a few new messages each round randomly: some parts of the protocol state machine may only become observable by using multiple messages from the same type but different subtypes. Take the Mealy machine in Figure 3.4, where *A1*, *A2* and *A3* are the same message type but different subtype. If we only add one new message of each type per round, the algorithm would never be able to find any of the states except for *q0*. This is because the only way to differentiate states is by the response for *B*, which only changes if all three subtypes of *A* are present. However after each run, only one *A* type message is left, because they appear to be identical in behaviour, so one of the two is removed, and only one is added next turn, ensuring no three subtypes are present in the same run. Even if we configure the algorithm to add two new messages per round, we have to get lucky to get the correct messages in the same turn.

To handle situations like these, the f_{gen} function needs to be improved. The improved version, f_{fbgen}, takes as input recorded network traffic, the Mealy machine produced in the previous round, f_{class} and produces a set of messages that should be added to Σ for the next round. At a high level, the function analyzes the network trace, and tries to identify *flows* that the Mealy machine would not be able to reproduce, then the messages in the flow which is not covered will be returned.

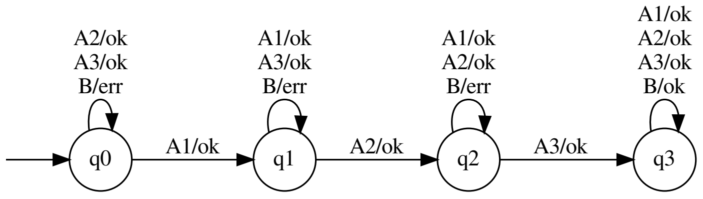

Figure 3.4. *Example Mealy machine MM_{pws}, to demonstrate problems with random message generation*

Definition 3.22 (Message flow)**.** *A message flow is an $(m_0, d_0), (m_1, d_1), \dots, (m_n, d_n)$ sequence, where $m_i \in S_{mtype}$ and $d_i \in \{request, response\}$. The sequence corresponds to a recorded session of communication, with m_i specifying the message type, and d_i specifying the direction of the message.*

First, the flows are extracted from the recorded network traffic by parsing and classifying them with f_{class}. Then, the Mealy machine is transformed into an FSM that accepts a message flow if the original Mealy machine could have produced it, and rejects it if it could not have. First off, the input alphabet of the FSM will be $\Sigma_{FSM} = S_{mtype} \times S_{mtype}$, so it will read a request and a response at the same time in each step from the message flow. The FSM can be produced from the Mealy machine through a few steps. First of all, the states of the FSM are initialized to be the same as the Mealy machine $Q_{FSM} = Q_{MM}$, the starting state is also the same in both machines, and all of these states will be accepting states. The transition function of the FSM is defined as $\delta_{FSM}\Big(q, \big[f_{gentype}(m), \lambda_{MM}(q,m)\big]\Big) = \delta_{MM}(q,m)$, for all $q \in Q_{MM}$ and $m \in S_{msubtype}$, where δ_{MM} and λ_{MM} are the transition and output function of the Mealy machine, while $f_{gentype} : S_{msubtype} \to S_{mtype}$ returns the message type a certain subtype belongs to. This basically means that the transitions of the Mealy machine are converted into the transitions of the FSM, by generalizing the input letter (which is a subtype in the Mealy machine) into a message type and merging the input and output letter. Turning subtypes into types is necessary, because the message flow has message types, since we cannot classify messages into subtypes. This way every message sequence, that the Mealy machine could have produced will be accepted, while if a certain sequence cannot be produced by the Mealy machine, the FSM will get stuck and reject it, because the input letter in question will not be defined in the given state. Notice, that by generalizing the subtypes into message types, we lose information, but as previously mentioned, this is unavoidable, and the resulting FSM in most cases still finds uncovered flows (i.e., flows that it rejects). Also, the resulting FSM is undeterministic and incomplete,

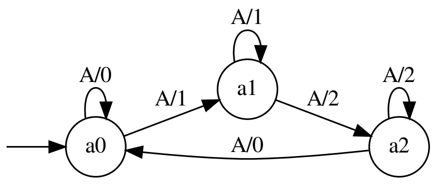

Figure 3.5. *M_A Mealy machine*

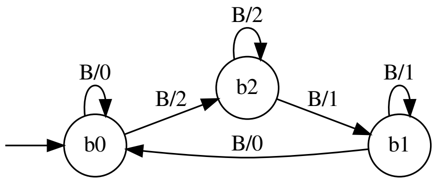

Figure 3.6. *M_B Mealy machine*

so it is usually a good idea to use the algorithms mentioned in Section 3.1 to fully define, determinize and minimize the FSM.

The f_{fbgen} function may also be used as a stopping criteria: as long as f_{fbgen} finds uncovered flows and returns new messages to be added, the algorithm continues, and once it does not find anything, the algorithm terminates.

3.2.5 Post-processing Mealy machines of protocols

Mealy machines can get very complicated in certain cases, which can make human analysis hard. One of the particular cases, where Mealy machines do not perform well, is handling cases where multiple, independent state spaces have to be tracked. Let us demonstrate this with an example: Let M_{comp} be the composite of two other Mealy machines: M_A and M_B, in Figure 3.5 and 3.6. If an input letter A is received the output letter should be produced using M_A, otherwise M_B should be used. The resulting M_{comp} machine can be seen in Figure 3.7.

As the example illustrates, M_{comb} becomes rather complicated, and it is not very straight forward to figure out what it does. The number of states in the combined machine is the product of the number of states in the machines that are being combined, since there has to be a new state for every combination of states. To find such independent component machines in Mealy machines and aid human analysis, let us define a *projection* of a Mealy as follows:

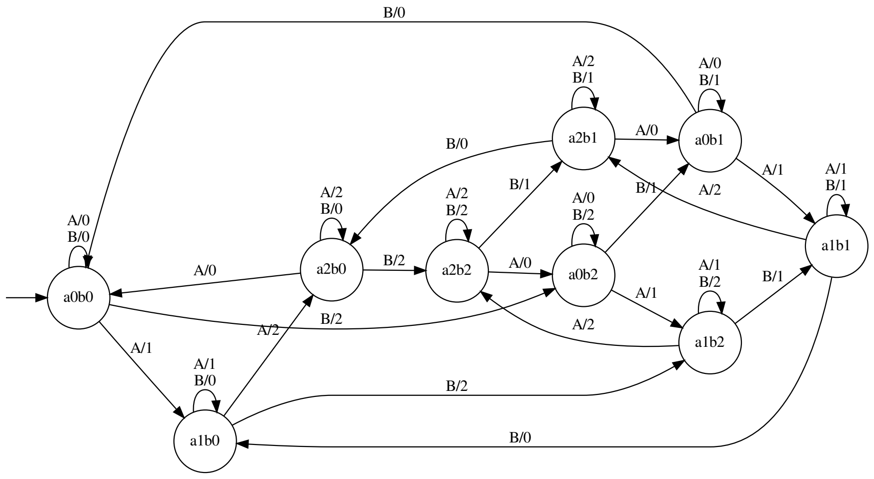

Figure 3.7. *M_{comp} Mealy machine*

Definition 3.23 (Projection)**.** *A projection of Mealy machine $M = (Q, \Sigma, \Delta, \delta, \lambda, q_0)$ on the set $S_p \subseteq \Sigma$, is an $M' = \big(Q, \Sigma, \Delta \cup \{l_{dontcare}\}, \delta, \lambda', q_0\big)$ Mealy machine that is equivalent to M in every aspect, except for the output function, which is redefined as $\lambda'(q, a) = l_{dontcare}$ for every $a \in \Sigma \setminus S_p$ and $\lambda'(q, a) = \lambda(q, a)$ for every $a \in S_p$.*

M' may not be minimal, since the output functions were changed to produce less information, so it should be minimized, which is what will actually reduce the number of states and make it easier to understand.

Using projections, we will create an algorithm, that takes a Mealy machine as input, and produces a number of Mealy machines, that together can produce the same output as the original. By using them together, I mean the following: every Mealy machine receives every input letter, and will produce some output. Some of these will be the $l_{dontcare}$ letter, and the rest will be the same letter, the letter that the original Mealy machine would have produced.

A trivial version of the algorithm is taking the projection for every letter in Σ, so the number of new Mealy machines would be $|\Sigma|$, each having an $S_p = \{l\}$ where $l \in \Sigma$. In fact, as long as every input letter in Σ is present in at least one of the S_p of the projections, the result of the algorithm is correct, since there will be always at least one Mealy machine, that can return the correct output for the input letter and not a $l_{dontcare}$. However, our goal is to aid comprehension and reduce complexity, so it is favourable to have as few Mealy machines in the output as possible, and with as few states as possible. To attempt this, we start out from the trivial version, and improve its results. For every projection, we try to add new input letters to S_p. If the new projection has the same amount of states as previously, we keep the new letter, otherwise we remove it. This way the number of input letters covered by a single projection increases, but the number of states do not. After trying every letter, we throw away any duplicate projections (so no pair of projections remain, for which $S_{p_i} = S_{p_j}$), and return the rest.

Chapter 4

Implementation

4.1 Base algorithm implementation

I created an implementation of the algorithms described in chapter 3 using the Python programming language. Python was chosen because it is well suited for fast prototyping, which was needed to validate my understanding of the ${L_M}^+$ algorithm and to be able to quickly test my own algorithms. One of the shortcoming of Python is that it is an interpreted language, which means that it performs slower than compiled languages do, but most of the time in the base algorithm is spent waiting on network communication, so that does not have too big an impact. Thanks to its popularity, Python also has a wide variety of libraries, both for academic purposes and practical applications. The following are the main dependencies that were used in the implementation:

- NetworkX[1] is a package for handling graphs, and includes multiple algorithms for working with and transforming graphs. In this implementation, it is used for representing Mealy machines and FSMs with directed graphs.
- fpdf[2] is a package that makes it possible to manipulate and create PDF files with Python. It is used for easing the debugging of the algorithms, by generating PDF files that contain the inner state of the algorithm at certain points of the run and the outputs.
- pyshark[3] is a wrapper around tshark, a command line version of the wireshark packet analyzer. It makes it possible to use the parsers and dissectors in wire-

[1]`https://networkx.org/` (last visited: 2020.11.30.)
[2]`https://pyfpdf.readthedocs.io/en/latest/` (last visited: 2020.11.30.)
[3]`https://kiminewt.github.io/pyshark/` (last visited: 2020.11.30.)

shark through python. In this implementation its mainly used to load PCAP files.

- docker MQTT[4] provides an API in python for accessing and manipulating the docker daemon. Docker containers are used for holding servers of the protocol under test, and this API makes it more straight forward to handle them.
- paho-mqtt[5] is a client implementation of the MQTT protocol, which was used to generate traffic for the algorithms as input.

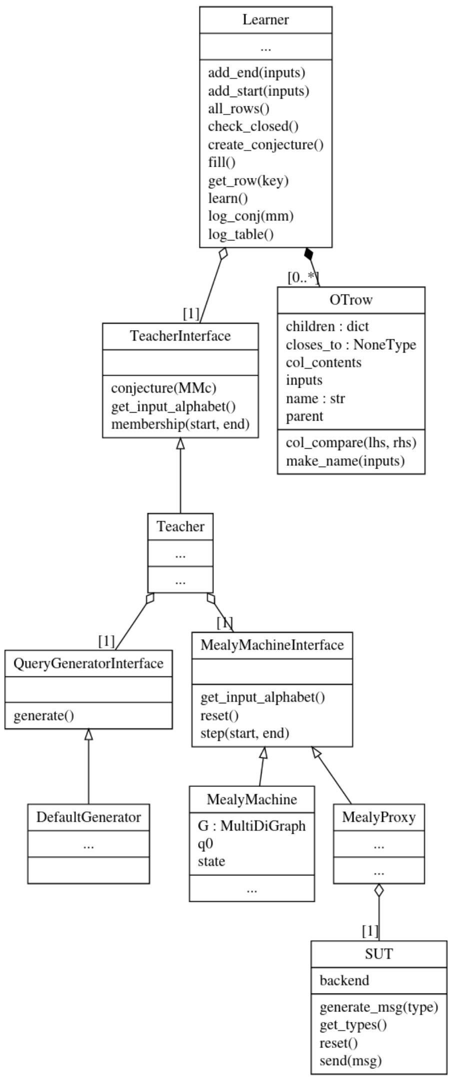

Figure 4.1. *Architecture of base algorithm implementation*

[4]https://pypi.org/project/docker/ (last visited: 2020.11.30.)
[5]https://pypi.org/project/paho-mqtt/ (last visited: 2020.11.30.)

```
+-------+---+---+-----+-------+
|       | b | a | a.a | b.a.a |
+-------+---+---+-----+-------+
|   *   | x | x | x.y | x.x.x |
|   a   | x | y | y.y | x.x.x |
|   b   | x | x | x.x | x.x.y |
|  a.b  | x | x | x.x | x.x.x |
|       |   |   |     |       |
|  a.a  | x | y | y.y | x.x.x |
|  b.b  | x | x | x.y | x.x.x |
|  b.a  | x | x | x.y | x.x.x |
| a.b.b | x | x | x.x | x.x.y |
| a.b.a | x | x | x.x | x.x.y |
+-------+---+---+-----+-------+
```

Listing 4.1. *Inner state of* ${L_M}^+$

4.1.1 Main architecture

The base algorithm is implemented based on [11], the class diagram of the base algorithm is in Figure 4.1. The two main components of the implementation are a *Learner* and a *Teacher* object. The learner contains the ${L_M}^+$implementation and uses the teacher to get information and storing the results of queries in a table made up of *OTrow*s. The teacher implements the required capabilities, such as output queries (membership), conjecture checking and querying the input alphabet. These are made available to the learner through function calls, and realized by calls to a generator object which generates random output queries for conjecture checking and by calls to an object with a *MealyMachineInterface* for output queries. This interface implements all functions that a Mealy machine needs to have, the main ones being stepping the machine, resetting the state, and listing the possible input letters. A simple output query is done, by sending a reset, and then stepping the machine with the requested input letters. The object implementing this interface can be a real Mealy machine, which was the case in early stages of the coding, where I needed to verify my implementation of the ${L_M}^+$algorithm. For testing, the program generates PDF files, which contain the inner state of the algorithm at various points, each page in the PDF corresponding to a snapshot of the inner state. The pages contain the table that holds the results of the queries, and the conjectured Mealy machine if applicable. The last page of a run, where the Mealy machine that is being learned is the one that is used in [11] as an example. The contents of the last page can be seen on Listing 4.1 and Figure 4.2. The implementation always names the starting state *, and the rest of the names of the states are taken from the first column of the table.

After the ${L_M}^+$implementation has been suffieciently tested, the Mealy machine was switched out to a *MealyProxy* that makes a real SUT appear as a Mealy machine to the teacher, by managing the state of the SUT and implementing and using the

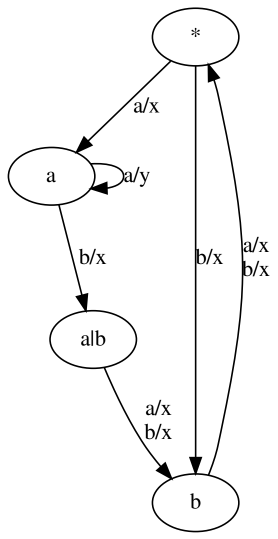

Figure 4.2. *Result Mealy for $L_M{}^+$testing*

three helper functions (f_{class}, f_{gen} and f_{upd}) defined in section 3.2.2 to query it. This is practical, because for targeting different protocols for APRE, one only needs to modify the proxy and the rest of the implementation can be left unchanged, allowing it to be handled by anyone as a black box, even if they do not understand how the algorithm or the implementation work, as long as they fulfil the requirement of being deterministic.

4.1.2 Implementation of the helper functions

All the protocols that were targeted were TCP based, so the f_{upd} function was essentially implemented by using standard TCP sockets, which take care of sequence numbers and other low level synchronisation. f_{class} was also simple, only a few bytes needed to be checked, as the protocols contained fields which explicitly defined the message type of the packet. A few more bytes were parsed that defined the length of the massage, allowing us to separate multiple messages, if sent in the same IP packet. Lastly, f_{gen} was constructed using f_{class} and recorded network traffic, as described in section 3.2.2. For this purpose, the recorded traffic is preprocessed with a separate script and stored in a JSON file, called *flows.json*. The script loads a PCAP file, and extracts the information that is needed by the base algorithm: the TCP payload is extracted from each packet in the capture, and given a unique ID. This way, if a payload is sent many times, it only has to be stored once, and can be referred to by its ID. Using *pyshark*, the TCP sessions in the capture are identified, and extracted:

```
{
    "flows": {
        "0": [
            [1337,"0"],
            [323232,"1"],
            [1337,"0"],
            [323232,"2"]
        ],
        "1": [
            [1337,"1"],
            [323232,"2"]
        ]
    },
    "payloads": {
        "0": "aGVsbG8gdGhlcmU=",
        "1": "Z2VuZXJhbCBrZW5vYmk=",
        "2": "eW91IGFyZSBhIGJvbGQgb25l"
    }
}
```

Listing 4.2. *Example flows.json file*

for every message in the flow, the ID of the payload and the destination port is recorded. This can be used to tell apart queries and responses, because servers and clients usually use different ports for communication, with the server usually having a fix port. An example output can be seen on listing 4.2, where there are three unique payloads (stored in base64 encoded form), from two TCP sessions, between the ports 1337 and 323232. Using this intermediate file allows us to keep the base algorithm free of TCP or other carrier specific code and saves storage space by deduplicating payloads.

4.1.3 Artificial message types

There is one last functionality that is fulfilled by the MealyProxy: to be able to model TCP based (and many other) protocols, some non-existent, artificial message types need to be defined. First, sometimes the server might silently accept a message and not respond. However, the algorithm should not just get stuck when this happens, so after some set timeout some kind of output must be defined for the Mealy machine. For cases like this, the *no response* message type is used, denoted as *noresp*. Second, TCP communication requires an established session, which can be torn down by either party with, or without signaling this. No communication will be possible, until a session is created, so the message type *sockconn* is used to direct the MealyProxy to initiate a new connection (and throw away the previous one, if there was any). The MealyProxy will always respond to *sockconn* with the same message type. When there is no valid socket, the proxy resonds with *noresp*. Additional, protocol specific artificial message types may be defined and implemented in MealyProxy if necessary.

4.2 Multiple round implementation

The extended version of the algorithm can be easily created by calling the base algorithm multiple times. The caller only has to manage the input alphabet: before every call, it requests new message subtypes from the MealyProxy and inserts them into the input alphabet. After each call returns with the result Mealy machine it needs to perform subtype deduplication on it and remove the unnecessary messages from the input alphabet. This is done by comparing pairs of message subtypes inside a message type one by one, and if their output function and transition function match for every state, one of them is removed. After this is done for every pair of message, only one of each subtype will remain.

These steps are repeated until the stopping createria is fulfilled. The simplest criteria is choosing a number after which many calls to the learning function without change in the size of the input alphabet the loop will exit, since the fact that the input alphabet is not growing means that only new members of the same subtypes are found and no new subtypes.

4.3 Caching query results

The biggest factor in the speed of the algorithm is the number of output queries made during the whole execution. This is because we have to set a timeout for how long we are willing to wait for bytes, which is the minimum time one query takes to finish, since we always want to receive all bytes sent, so even if the server instantly responds, we have to make sure it does not want to send any more data. Since the environment needs to be deterministic, we can store the result of previous queries and use them to answer without actually sending and receiving anything. Repeated queries for the same input can happen for various reasons: having multiple runs of the ${L_M}^+$algorithm with the same target, even if with slightly different input alphabets, the start of the run will contain overlaps where queries can be reused. But even within the same run, if two queries are made where the first starts with the second (for example the first is aab and the second is aa), the second can be answered by taking the first part of the output of the first query (with the previous example that would be the first two output letters). Using a cache like this presents a nice opportunity to double check the implementation for errors and the environment for inconsistent or undeterministic behaviour. To continue the previous example, if the same two requests are made, but in reverse order, it can be now checked if the SUT responds the same way to the first part of the query.

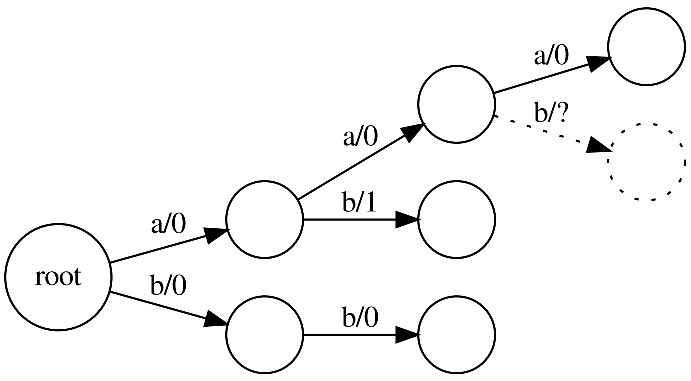

Figure 4.3. *Example query cache*

The cache functionality is implemented inside the teacher output query, so everything that makes queries to the SUT will benefit from the caching automatically (including the conjecture checking). A tree structure is used for the implementation, each node in the tree holds the output letter corresponding to the input letter, if the input letters along the route to the root of the tree were input previously. On the example in Figure 4.3 the edges are labeled with the corresponding input letter and output letter to that path and the dotted edges represent queries that have not been made yet. So if an output query came for *ab*, it could be answered from cache, as *01*, but if a query for *aab* came, the whole query would have to be executed against the SUT, but the outputs for *aa* along the existing nodes could be verified and the new output would be inserted as a new node.

4.4 Optimizing the order of queries

As a result of how the ${L_M}^+$ algorithm works, output queries are made in bulk, then based on the results of those queries, either more queries are made (also in bulk), or a conjecture is formed. With the way the cache is implemented, as described in section 4.3, it is favourable to make longer queries first, to make it more likely to have cache hits. The solution is sorting queries in descending order by length, before passing them one by one to the teacher.

4.5 Parallelizing

As already mentioned in section 4.3, a very big part of the runtime of the algorithm is taken up entirely by waiting for network replies. This makes parallelization a very good way to increase the efficiency of the implementation: if we can wait for more than one query at the same moment, a lot of time can be saved. Multithreaded programs are not always the best idea to implement in Python, because of the global interpreter lock (GIL), which prevents multiple threads of Python code to run at the same time, ensuring thread safety. As a result there is usually no performance gain when parallelizing algorithms implemented in Python, as all the code will just run on the same CPU/thread anyways. However, in this case, when the *recv* function of a socket is waiting for input, that thread will just go to sleep, and allow other threads to execute, allowing us to reach the timeout, while also working on other operations and not wasting as much time. Depending on the number of available server implementations that can be queried at the same time, there may be several threads with sockets waiting for timeout at the same time. And as a bonus, the GIL saves me from most of the headaches concerning race conditions and deadlocks that are associated with multithreaded implementations.

To allow for parallel execution, the input query function of the teacher can be called in either blocking or non-blocking mode. In non-blocking mode, if the query is already cached, it will be immediately answered, if it is not, an available worker is assigned to it (making it unavailable). A thread is created that will perform the query, and an entry will be made into the cache, to signal that it is already being worked on and by which thread and a placeholder value is returned. If there is no worker available, the call will block until one becomes available. Once the thread is finished with the query, it switches out the placeholder values for the real output letters in the cache, releases the worker and terminates. Other than avoiding starting multiple threads for the same output queries, having the handle of the thread in the cache is useful, because it allows whoever needs the value that will be written there (for example the blocking version of the output query) to efficiently wait for the query to be finished by waiting for the thread to terminate.

So now whenever a bulk of output queries need to be made, the caller can call the function with every query in non-blocking mode first, and then in blocking mode. The MealyProxy does have to handle the workers, by having an extra parameter for the worker ID in its functions and forwarding the messages to the appropriate SUT. A simple concurrent queue for holding which workers are currently not assigned to any task is also necessary. Whenever a free worker is needed, it is popped off from the head of the queue, and when one is released it is put back into the end.

4.6 Shortening counterexamples

Counterexamples are found by randomly generated input sequences, which can result in very long counterexamples. This directly influences the number of messages that the base algorithm has to send, the longer the counterexamples, the more time is spent waiting for replies. These randomly generated counterexamples could in some cases be shortened: for example if at any point during the sequence both the real and the (falsely) conjectured Mealy machine returns to their respective starting state, the part up to that point can be safely discarded from the sequence and it will still remain a counterexample. Other cases can also be possible where the counterexample can be shortened. To find the shortest version of a counterexample, starting from the end of the sequence increasing in length each subsequence of the original is tested, and the first time that the SUT and the conjecture give different results, we have found the shortened version. To execute the search faster, a binary searching approach can be taken: first the start of the sequence is moved to the middle of the original sequence and tested. If it remains a counterexample the start is moved to the right into the middle, otherwise to the left, and so on, until the exact point is found where the sequence starts being a counterexample. Unfortunately, this version is not guaranteed to always find the shortest version, however if there is a message that takes both machines to the same state (a one length homing sequence), it can work really well. In most cases, *sockconn* is like that, which is why I implemented the binary search version.

Chapter 5

Evaluation

5.1 Testing with a toy protocol

To test the viability of running the base algorithm multiple times while expanding and deduplicating the input alphabet, a simple toy protocol was created. The read and write message types have two subtypes each: one with valid target address (*get* and *write*) and one with bad target address (*bad_ get* and *bad_ write*). The valid get messages can be used both in *BASE* and *WRITE* mode, but the write message can only be used in *WRITE* mode. The full state machine of the toy protocol can be seen in Figure 5.1.

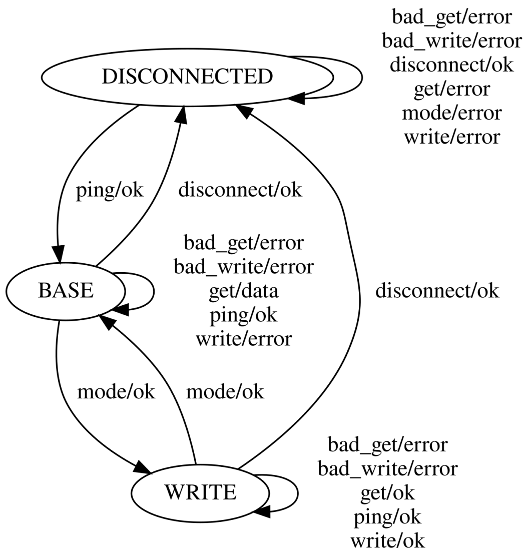

Figure 5.1. *Mealy machine of the toy protocol*

In this test, no network communication was involved, the protocol was implemented entirely in the MealyProxy, the goal being the verification of the implementation of the multiple round extentions. f_{gen} was implemented simply by making a random choice of whether the generated *get* and *write* is valid or not. Each round, one new message from each subtype was generated. The output after the first round can be seen in Figure 5.2. Since only one message is present from each message type, deduplication has no effect. Both the generated get and write commands happened to come with invalid target addresses, so only two states are discovered, since the bad versions always return with an error, there is no way to tell the *BASE* and *WRITE* states apart.

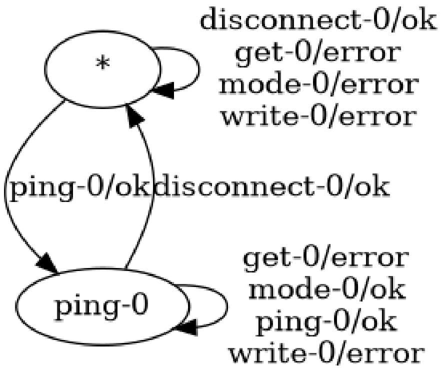

Figure 5.2. *Output for toy protocol after the first round*

After the second run, enough valid messages get generated to differentiate all three states of the protocol. In Figure 5.3 the output before deduplication can be seen, while Figure 5.3 shows it after deduplication. Before deduplication, there are two messages from each message type, but after deduplication only the *get* type remains duplicated, so we found the two get subtypes.

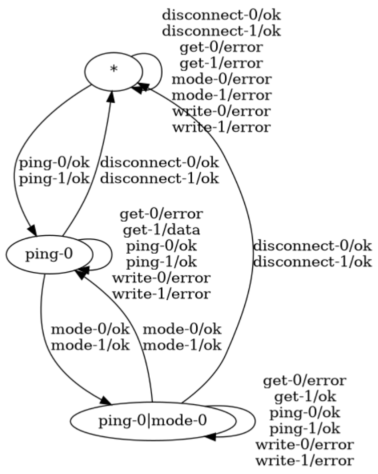

Figure 5.3. *Output for toy protocol after the second round before deduplication*

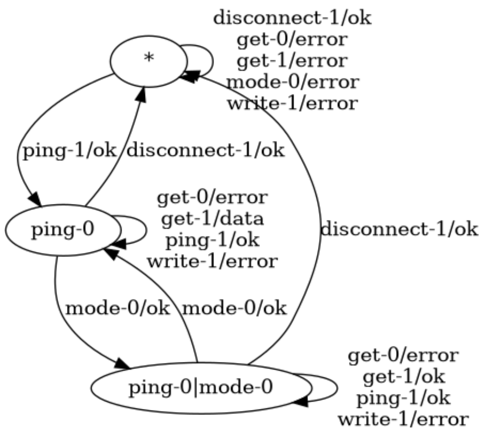

Figure 5.4. *Output for toy protocol after the second round with deduplication*

And finally, after the third round all subtypes are found, as shown in Figures 5.5 and

5.6. The deduplicated machine is equivalent to the original protocol state machine. A number of rounds were run after this, but nothing changed, and the algorithm stopped as the stopping criteria was fulfilled.

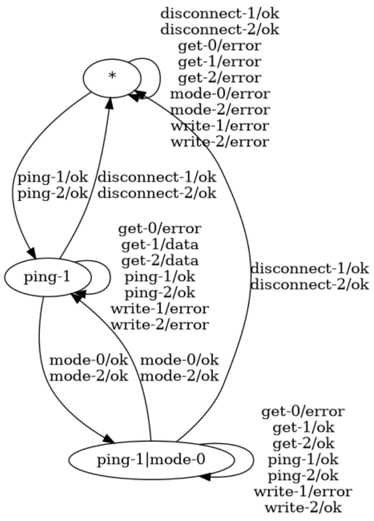

Figure 5.5. *Output for toy protocol after the third round before deduplication*

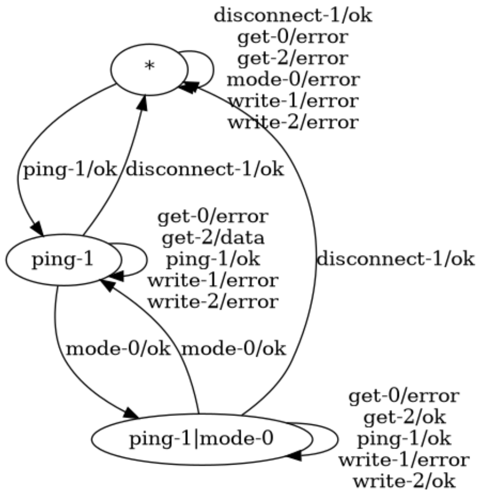

Figure 5.6. *Output for toy protocol after the third round with deduplication*

5.2 Modbus

Modbus is a widely known protocol, mainly used in industrial settings, to read and write values from and to programmable logic controllers (PLC). It is a fairly simple protocol, and the specifications are open and freely available. These properties make it ideal for evaluating my implementation. For generating traffic for recording and serving as a SUT, a custom implementation of the Modbus protocol (both client and server) was used, created by Gergő Ládi at CrySyS laboratory. In addition to the normal messages in Modbus, an additional message was added that allows resetting the state of the server remotely.

Traffic was generated by using the client and recorded into a PCAP file, which converted into flows using the script described in section 4.1.2. Every Modbus packet starts with a fix 8 byte header, which makes parsing easy: the f_{class} function uses only one byte for determining the message type, and uses the a few more bytes to determine the length of the packet. f_{gen} was realized by using f_{class} and selecting from the messages in the captured flows, as described in section 3.2.2. No special care needed to be taken regarding f_{upd}, except correctly using socket communication. The artificial message types *noresp* and *sockconn* were also used. The output of the algorithm for Modbus can be seen in Figure 5.7. The input letters are of the form *messagetype-messageid*, where the message types are simply the eighth byte of the message. The result is that Modbus has a rather simple state machine, with only two states, that correspond to whether the client is connected or not.

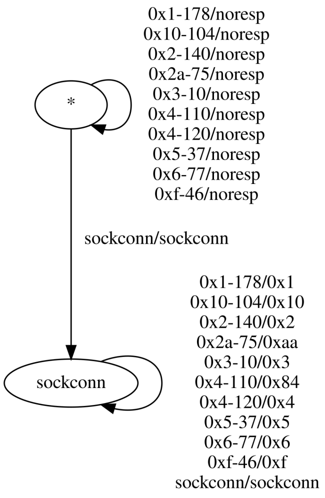

Figure 5.7. *Modbus protocol state machine generated by the algorithm*

5.3 MQTT

MQTT is a relatively simple, publish-subscribe based protocol, where clients can subscribe to channels, and messages published by clients to channels will be sent to all clients that are subscribed to that channel. It is mainly used in industrial and IoT settings, thanks to its simplicity and open specifications. In addition, three Quality of Service (QoS) levels can be set: at QoS 0, messages are simply sent and nothing else is done, while on QoS level 1 and 2 increasingly complex handshakes are completed to publish messages. This is useful for setting the balance in resource constrained systems between error correction and message length and implementation complexity. For APRE it is nice to have settings with different complexity levels so the algorithm can be tested at different complexity levels.

MQTT also has a wide open source software support, which made it easy to set up a SUT environment. For the server implementation Eclipse Mosquitto[1] was used, and for the client side the Eclipse Paho™ MQTT Python Client[2] library was used to generate sample traffic in a custom Python script. To ease installation and later be able to easily instantiate multiple servers at the same time and easily reset them, the Mosquitto server was used inside a docker container, with the *eclipse-mosquitto* image[3] from Dockerhub. The Python script performs a sequence of randomly generated

[1]`https://github.com/eclipse/mosquitto` (last visited: 2020.11.30)
[2]`https://github.com/eclipse/paho.mqtt.python` (last visited: 2020.11.30)
[3]`https://hub.docker.com/_/eclipse-mosquitto` (last visited: 2020.11.30)

actions available through the API of the Paho MQTT library, and the generated traffic is recorded. Unfortunately, if the commands are called too quickly in succession more than one message might be sent in a single IP packet, which makes converting into flows near impossible, so *sleep* commands are placed between them, but sometimes even this is not enough. This is because the Linux kernel contains optimizations for sending TCP traffic, using Nagle's algorithm, which bundles multiple messages into the same packet, so the overhead of the various IP and TCP headers does not decrease efficiency too much. To counteract this, I set an artificial latency on localhost communications and enabled low latency mode TCP with the commands on Listing 5.1:

```
sudo bash -c "echo 1 > /proc/sys/net/ipv4/tcp_low_latency" # enable low latency TCP
sudo tc qdisc add dev lo root handle 1:0 netem delay 10msec # localhost latency
```

Listing 5.1. *Commands for ensuring no packets are merged*

With network traces available, f_{gen} can be constructed the usual way, and f_{fbgen} can also be used. In addition, the QoS level used to generate the sample traffic will influence what QoS the protocol will be analyzed with. f_{class} uses the first four bits in the constant header of the MQTT messages to categorize messages, since according to the documentation these bits determine the packet type. The names of the various packet types were also included in the classifier, so the produced Mealy machines are easier to understand. A few more bytes are also parsed to determine the length of the messages. f_{upd} is handled in the same way as with Modbus, using simple sockets takes care of communication synchronization.

Docker allows for easily resetting the state of the mosquitto server, by simply restarting the whole container. To avoid any state being stored on disk, the container is configured to have read-only storage. The docker containers of mosquitto are handled by the MealyProxy through the docker Python SDK. Multiple instances of the container are started with the servers listening port forwarded to different ports, and each container is assigned to a worker to allow multithreaded execution. In order to simulate other clients, a third artificial message type is defined alongside *sockconn* and *noresp*: this message type (*pubtrigger*) was created in order to simulate message publications made by other clients connected to the same server. When the MealyProxy receives such a message, it publishes a message to a certain channel using the Paho MQTT library, and listens for any messages on the socket currently defined in the proxy (if there is one).

The tests that were executed were configured by the different setups that the sample network traffic was generated with. In every setup, the server was configured with a few users, and anonymous connections (ones, where passwords are not necessary)

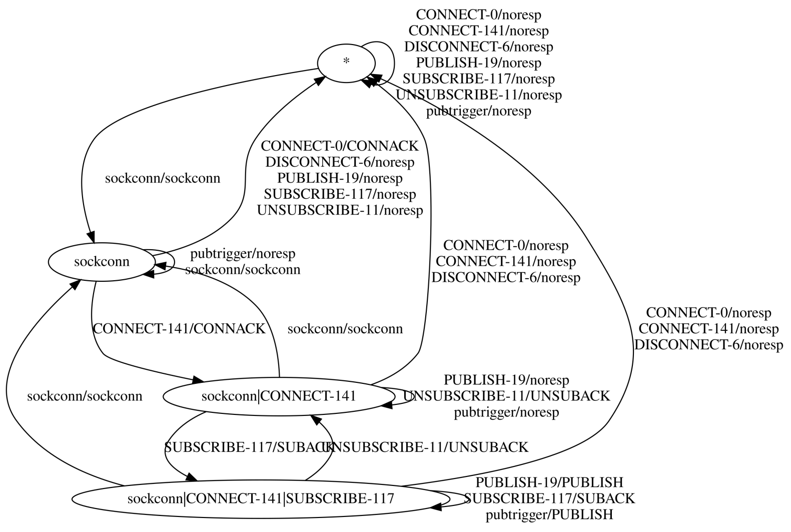

Figure 5.8. *Result Mealy machine for MQTT with a single channel and QoS 0*

were disabled. There were always a number of channels, messages and QoS levels that the client script could randomly choose from, to subscribe and publish. The simplest test involved only a single channel and only QoS 0. The result is in Figure 5.8, where it is clearly visible, how the server stores that state of the client: when the client subscribes, it moves to a state where publish messages are sent to the client, when it unsubscribes it moves to one, where they are not sent. The only message type where subtypes are found is *CONNECT*, where *CONNECT-0* is the one with invalid credentials and *CONNECT-141* is the one with correct credentials. Another interesting detail that can be read from the figure, is that the *CONNECT* message has to be sent exactly after establishing the TCP session, if anything else is sent first, or if a *CONNECT* message is sent later in the session, the server instantly tears down the connection without sending any error messages.

In the next test, two channels were used, and as one would expect, there is a state for every combination subscriptions of channels in Figure 5.9. This way, we have in two more states resulting in a total of four ($= 2 * 2$) states used for managing subscription state. In this run we also have subtypes for *SUBSCRIBE*, *UNSUBSCRIBE* and *PUBLISH* as well. With a bit of analysis it can be deduced that the triplet of *SUBSCRIBE-176*, *UNSUBSCRIBE-132*, *PUBLISH-10* target one of the channels and *SUBSCRIBE-129*, *UNSUBSCRIBE-153*, *PUBLISH-21* target the other. In addition, the second trio belongs to the same channel as the one where *pubtrigger* publishes.

With three channels the number of states needed to handle subscriptions increases to eight, where the general formulat is 2^n for the number of states needed, where n is the number of channels. As the number of channels increase the Mealy machine is

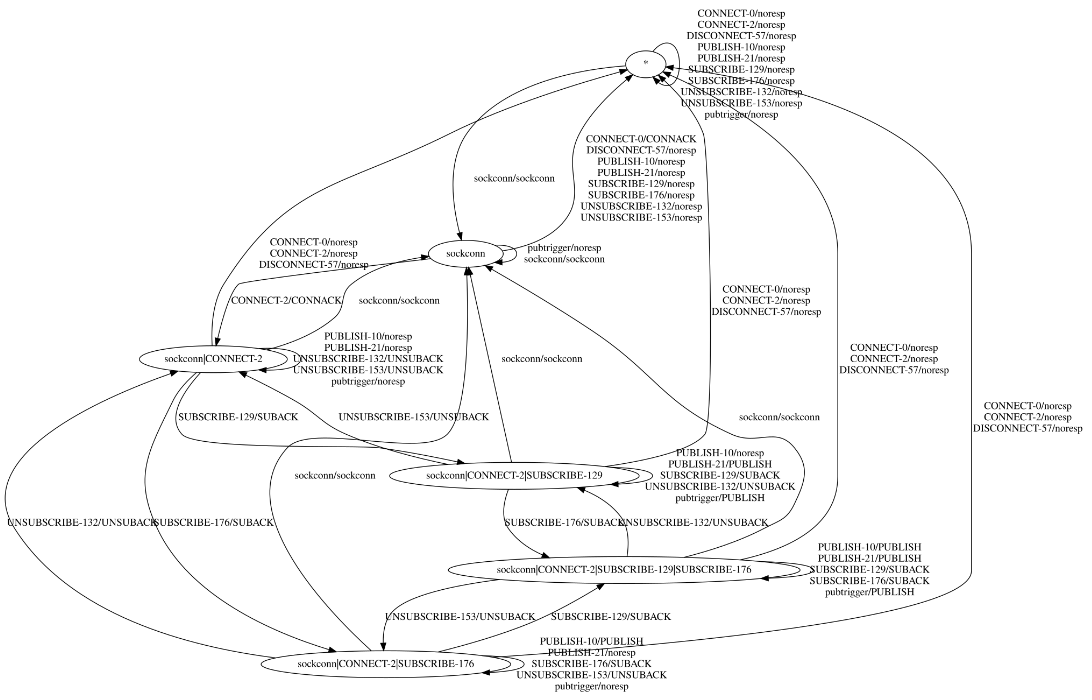

Figure 5.9. *Result Mealy machine for MQTT with two channels and QoS 0*

increasingly complicated, however subscriptions to various channels are completely independent. Mealy machine projections defined in section 3.2.5 are meant to simplify Mealy machines like this. The result of the algorithm with three channels and automatic projection after is in Figure 5.10. Just by looking at the shapes, it is easy to see that the bottom three Mealy machines serve the same function, and after realizing that they are responsible for handling subscriptions they can be used to easily collect which *PUBLISH*, *SUBSCRIBE* and *UNSUBSCRIBE* messages belong to the same channel. It can also be seen on the Mealy machine with three states, that if we do not care about the responses to *PUBLISH* messages, then the responses to the rest of the messages can be predicted with only three states. Likewise two states is enough if we only care about *CONNECT*, *DISCONNECT* and *sockconn* queries; and finally the responses for *DISCONNECT* and *sockconn* queries are completely independent of the state of the protocol state machine.

Tests with QoS 1 did not yield that much different results, the only change was, that for *PUBLISH* messages, there was always a *PUBACK* response, which means when the client is also subscribed to the correct channel, two messages come as a response to one. To be able to work with this, new message types are defined dynamically that represent that those two messages (in this case *PUBACK* and *PUBLISH*) were both received as a response. This is possible, because only the input alphabet is required to be known beforehand by ${L_M}^+$ so the output alphabet can easily be expanded at any time.

On the other hand, QoS 2 produced bigger and more complicated Mealy machines, and unfortunately simple projections were not very effective at simplifying them. Because of this human analysis was not successful either, so more post-processing techniques will be needed in the future to be able to work with QoS 2.

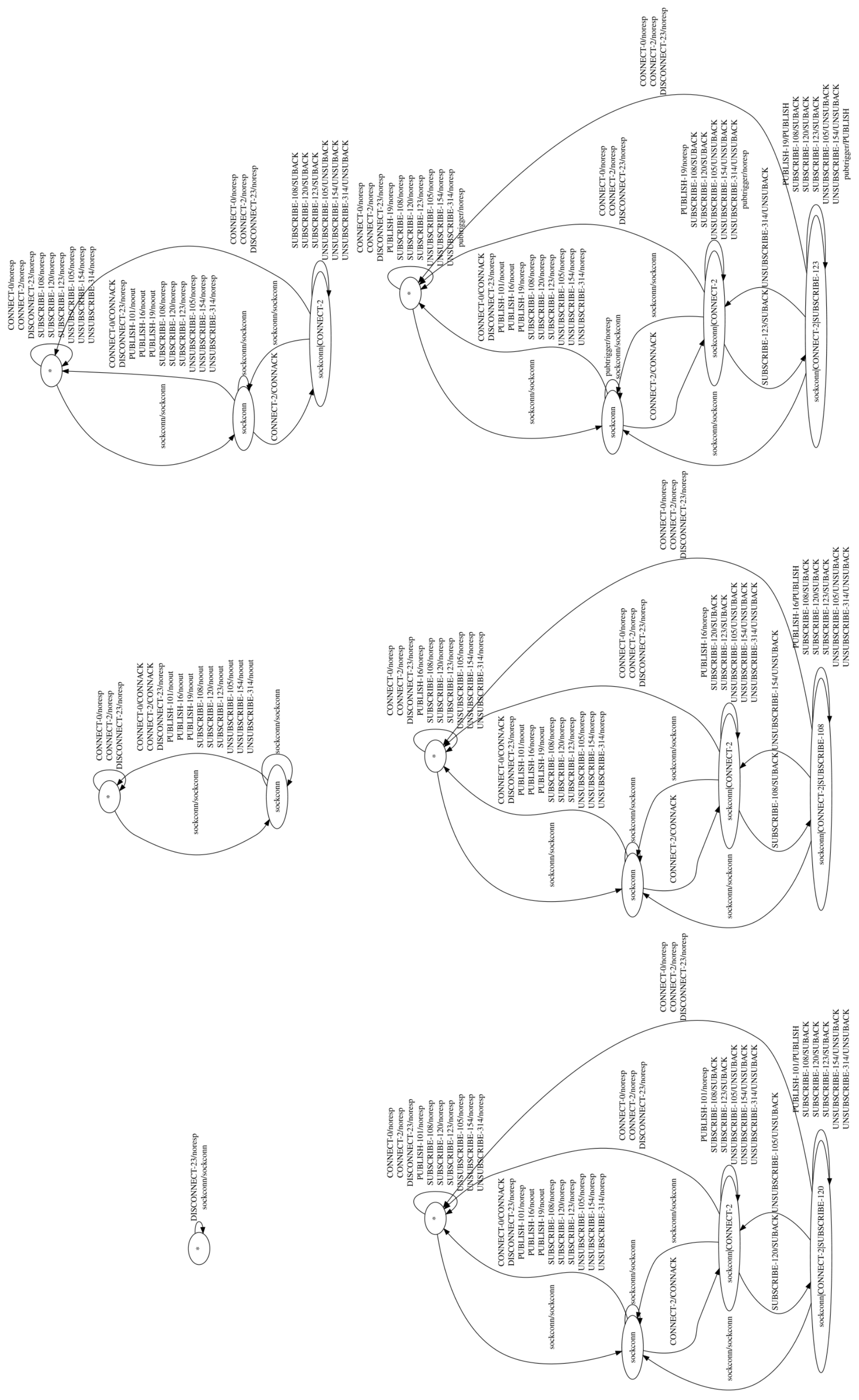

Figure 5.10. *Projections of the Mealy machine of MQTT with three channels*

Chapter 6

Conclusion

In this thesis a model for representing the state machine of a protocol with Mealy machines is introduced. Assuming some prior, at least partial knowledge of the syntax of the messages in the protocol and recorded communication, an application and extension of the ${L_M}^+$algorithm is created with multiple optimizations to automatically reverse engineer the state machine of the protocol. An implementation was created and successfully evaluated on a test protocol and two real world protocols: MODBUS and MQTT.

All results except the highest level QoS MQTT test were evaluated, and by my understanding correctly represented the protocol that was being reversed. A post-processing algorithm was also created, aiming to simplify the result Mealy machine and ease understanding of the generated models. The effectivity of this method is demonstrated through a test with MQTT on QoS 0 with three channels, where it simplified the representation of the logic of handling subscriptions. Unfortunately this method was not effective at simplifying the result Mealy machines for QoS 2, and as a result I was not able to manually verify.

Although the model presented in [13] is not exactly for protocols, assuming a similar process for APRE, I believe the toolset presented here is suitable for the steps *overview* and *focused experimentation* through the manipulation of the granularity of message types, and the set of messages in the captured traffic.

As future work, other automata decomposition techniques should be explored, to allow simplification of more complex Mealy machines. This would make it possible to understand more complicated Mealy machines and the protocols corresponding to these Mealy machines.

Acknowledgment

I would like to thank my thesis advisor Dr. Levente Buttyán and everyone else who helped me over the years in the CrySyS laboratory at the Budapest University of Technology and Economics, and my colleagues from Ukatemi Technologies. They provided me with valuable advice, guidance and encouragement.

Listings

List of Figures

Acronyms

APRE Automatic Protocol Reverse Engineering. 4, 5, 7–9, 21, 33, 48

FSM Finite State Machine. 15–21, 26, 27, 30

GIL Global Interpreter Lock. 37

IDS Intrusion Detection Systems. 7

IoT Internet of Things. 6, 42

JSON JavaScript Object Notation. 33

MQTT Message Queuing Telemetry Transport. 2, 4, 5, 31, 42–45, 47, 48, 51

PAM Partitioning Around Medoids. 10

PCAP Packet Capture. 31, 33, 41

PDF Portable Document Format. 30, 32

PLC Programmable Logic Controller. 41

QoS Quality of Service. 42–46, 48, 51

SMB Microsoft Server Message Block. 6, 7

SUT System Under Test. 23, 32, 35–38, 41, 42

Bibliography

[1] Dana Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75(2):87–106, Nov 1987.

[2] João Antunes, Nuno Ferreira Neves, and Paulo Verissimo. ReverX: Reverse engineering of protocols. *12th International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT)*, 2011.

[3] Juan Caballero, Pongsin Poosankam, Christian Kreibich, and Dawn Song. Dispatcher: Enabling active botnet infiltration using automatic protocol reverse-engineering. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 621–634, 2009.

[4] Georg Chalupar, Stefan Peherstorfer, Erik Poll, and Joeri De Ruiter. Automated reverse engineering using Lego®. In *8th USENIX Workshop on Offensive Technologies (WOOT 14)*, 2014.

[5] Chia Yuan Cho, Domagoj Babi ć, Eui Chul Richard Shin, and Dawn Song. Inference and analysis of formal models of botnet command and control protocols. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 426–439, 2010.

[6] Paolo Milani Comparetti, Gilbert Wondracek, Christopher Kruegel, and Engin Kirda. Prospex: Protocol specification extraction. *30th IEEE Symposium on Security and Privacy*, pages 110–125, 2009.

[7] Weidong Cui, Jayanthkumar Kannan, and Helen J. Wang. Discoverer: Automatic protocol reverse engineering from network traces. *SS'07 Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium*, 2007.

[8] Zhiqiang Lin, Xuxian Jiang, Dongyan Xu, and Xiangyu Zhang. Automatic protocol format reverse engineering through context-aware monitored execution. In *NDSS*, volume 8, pages 1–15, 2008.

[9] Gergő Ládi, Levente Buttyán, and Tamás Holczer. GrAMeFFSI: Graph analysis based message format and field semantics inference for binary protocols using recorded network traffic. *Infocommunications Journal*, 12(2):25–33, August 2020.

[10] John Narayan, Sandeep K. Shukla, and T. Charles Clancy. A survey of automatic protocol reverse engineering tools. 48(3), December 2015.

[11] Muzammil Shahbaz and Roland Groz. Inferring Mealy Machines. In *International Symposium on Formal Methods*, pages 207–222. Springer, 2009.

[12] Gábor Székely, Gergõ Ládi, Tamas Holczer, and Levente Buttyán. Towards reverse engineering protocol state machines. In *The 12th Conference of PhD Students in Computer Science - Volume of short papers*, pages 70–73, 2020.

[13] Daniel Votipka, Seth Rabin, Kristopher Micinski, Jeffrey S Foster, and Michelle L Mazurek. An observational investigation of reverse engineers' processes. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1875–1892, 2020.

[14] Y. Wang, Xiaochun Yun, M. Z. Shafiq, L. Wang, A. X. Liu, Z. Zhang, D. Yao, Y. Zhang, and L. Guo. A semantics aware approach to automated reverse engineering unknown protocols. *20th IEEE International Conference on Network Protocols (ICNP)*, pages 1–10, 2012.

[15] Yipeng Wang, Xingjian Li, Jiao Meng, Yong Zhao, Zhibin Zhang, and Li Guo. Biprominer: Automatic mining of binary protocol features. *12th International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT)*, pages 179–184, 2011.

[16] Yipeng Wang, Zhibin Zhang, Danfeng Yao, Buyun Qu, and Li Guo. Inferring protocol state machine from network traces: A probabilistic approach. *ACNS 2011: Applied Cryptography and Network Security*, pages 1–18, 2011.

[17] Zhi Wang, Xuxian Jiang, Weidong Cui, Xinyuan Wang, and Mike Grace. Reformat: Automatic reverse engineering of encrypted messages. In *European Symposium on Research in Computer Security*, pages 200–215. Springer, 2009.

[18] Z. Zhang, Q. Wen, and W. Tang. Mining protocol state machines by interactive grammar inference. In *2012 Third International Conference on Digital Manufacturing Automation*, pages 524–527, 2012.