Acta Cybernetica **25** (2021) 517–535.

Protocol State Machine Reverse Engineering with a Teaching–Learning Approach*

Gábor Székely[abc], Gergő Ládi[ade], Tamás Holczer[af], and Levente Buttyán[ag]

Abstract

In this work, we propose a novel solution to the problem of inferring the state machine of an unknown protocol. We extend and improve prior results on inferring Mealy machines, and present a new algorithm that accesses and interacts with a networked system that runs the unknown protocol in order to infer the Mealy machine representing the protocol's state machine. To demonstrate the viability of our approach, we provide an implementation and illustrate the operation of our algorithm on a simple example protocol, as well as on two real-world protocols, Modbus and MQTT.

Keywords: automated protocol reverse engineering, state machines, Mealy machines

1 Introduction

In today's world, IT systems are seldom standalone and monolithic. Each system is comprised of up to several tens of parts or modules that somehow communicate with each other, usually via a network. The rules of communication, i.e. the formats of the messages and the possible valid sequences of messages are established by protocols that are defined in specifications. Many systems use proprietary or closed protocols whose specifications are not made publicly available. Examples typically include industrial control systems (ICS) and in-vehicle embedded networks.

*The research presented in this paper has been supported by the European Union, co-financed by the European Social Fund (EFOP-3.6.2-16-2017-00013), the Hungarian National Research, Development and Innovation Fund (NKFIH, project no. 2017-1.3.1-VKE-2017-00029), and the IAEA (CRP-J02008, contract no. 20629).

[a]Laboratory of Cryptography and System Security, Department of Networked Systems and Services, Budapest University of Technology and Economics, Hungary

[b]Ukatemi Technologies

[c]E-mail: gabor.szekely@ukatemi.com, ORCID: 0000-0001-6148-3948

[d]BME Balatonfüred Student Research Group, Hungary

[e]E-mail: gergo.ladi@crysys.hu, ORCID: 0000-0002-0318-2175

[f]E-mail: holczer@crysys.hu, ORCID: 0000-0003-0953-5397

[g]E-mail: buttyan@crysys.hu, ORCID: 0000-0003-4233-2559

DOI: 10.14232/actacyb.288213

However, it would often be largely beneficial to have at least an approximate idea of how these closed protocols work. For instance, with this knowledge, it would be possible to build network anomaly detection tools that detect potential cyberattacks against industrial systems, design more specialized honeypots, detect malicious or malfunctioning components in in-vehicle networks, or build new components that can be more easily integrated into existing systems.

One may attempt to reconstruct the specification of an unknown protocol by applying various protocol reverse engineering methods. The goal of such methods is two-fold: first, the type and format of the messages used by the protocol need to be understood, second, this information may be used to recover the state machine of the protocol. The process of reverse engineering a protocol is often a tedious task; therefore, some degree of automation is required to make it practical.

In a previous paper [14], we focused on the problem of determining the message types and message formats of undocumented binary protocols. We developed a tool that can process captured network traffic containing messages of a protocol, then retrieve the identified message types as well as the semantics of the message fields for the different message types. In this paper, we address the problem of inferring the state machine of a previously unknown protocol.

Our method is based on prior work on inferring Mealy machines, in particular, on the work of Shahbaz and Groz [20]. We take their Mealy machine inference algorithm and extend it with elements that make it possible to use their conceptual results in practice so as to reverse engineer the state machine of real-world protocols in an automated fashion. This method requires access to and interaction with a system that runs the unknown protocol. In addition, we assume that the message types and formats have already been reverse engineered, e.g., by using our previously published method. We process and use previously recorded network traces as well as data obtained from this system, efficiently and intelligently choosing input for the above-mentioned Mealy machine inference algorithm.

2 Related work

The principles of protocol reverse engineering date back to the 1950s, where reverse engineering was typically used for fault analysis, and it involved analyzing and understanding electrical circuits that implemented a finite state machine representing a protocol [15]. With computers and computerized accessories becoming more and more widespread around the turn of the century, the number of network applications, thus the number of network protocols increased. Some of these were proprietary with no available documentation, meaning that they had to be reverse engineered in order to develop compatible applications.

The first well-known project that aimed at restoring the specifications of such a protocol was the Samba Project (2003) [1] that has taken 12 years to finish. The exact methods they used are not detailed, but it is known that they sniffed network traffic and employed random probing to identify message types and semantics. The Samba Project was soon followed by M. A. Beddoe's Protocol Informatics Project

[5] in 2004, which used bioinformatical algorithms on network traces to infer the message types of the text-based protocol HTTP. RolePlayer (2006) [12], Discoverer (2007) [11], Biprominer (2011) [22], ReverX (2011) [3], ProDecoder (2012) [21], and AutoReEngine (2013) [17] soon followed, all of which relied solely on network traffic.

The early works typically focused on message type and message format inference. They did not put much emphasis on field semantics inference, nor did they attempt to recover the protocol state machine. Those that tried to infer field semantics did not achieve significant results – Discoverer admits to achieving between 30-40% accuracy [11], and not even Netzob exceeds 50% [7]. FieldHunter (2015) [6] was the first to achieve over 80% accuracy on semantics. In a previous paper [14], we presented GrAMeFFSI (2020) that may achieve over 90% accuracy on binary protocols if high-quality network captures are available.

While most algorithms aimed at reversing both text-based and binary protocols, some specialized in one or the other, usually achieving better performance metrics compared to the more general solutions of their time. Biprominer, as its name suggests, targeted binary protocols, as did GrAMeFFSI, while ReverX targeted text-based protocols. The methods employed vary – RolePlayer and Discoverer rely on sequence alignment, Biprominer and AutoReEngine leverage data mining approaches, ProDecoder makes use of natural language processing algorithms, while GrAMeFFSI employs graph analysis.

An alternative to network traffic analysis based protocol inference is binary analysis based inference. Such methods often rely on dynamic taint analysis, marking sections of code in the memory of the binary being executed that are hit when processing or responding to a given message, then making assumptions about the message formats and semantics based on what and how was marked. It has been proven [19] that binary analysis based approaches can achieve better results; however, purely traffic analysis based approaches are also important as binaries may not always be at our disposal, and legal agreements may prevent us from analyzing or reverse engineering these.

In order to reverse engineer the protocol state machine, at least a partial understanding, a classification of the message types is needed. State machine inference methods must either produce the message classes themselves, or rely on existing message format inference methods, perhaps in a slightly modified manner. So far, there have been fewer attempts to reconstruct protocol state machines than to determine message types and semantics [13]. The majority of the network trace based solutions are passive, meaning that only recorded traffic is used as input, and no live systems (running original protocol implementations) are contacted.

ScriptGen (2005) [16] was the first notable method for state machine inference. It uses the Needleman-Wunsch sequence alignment algorithm (similarly to Beddoe's previously mentioned project) along with micro- and macroclustering to build a protocol state machine from captured network traffic. The state machine is then used to formulate responses for a honeypot. It was later followed by Cho et al.'s work on reverse engineering the protocol of the MegaD botnet (2010) [8] that leveraged and optimized Angluin's L^* algorithm [2] to infer a Mealy machine

based on network traces and live queries. They reached between 96% and 99% accuracy on various protocols. Another important result was Veritas (2011) [23] that employs statistical analysis on captured network traffic to build probabilistic protocol state machines that are claimed to be 92% accurate on average. There also exist approaches that rely on program execution instead of network traces, the most significant ones being Prospex (2009) [10] and MACE (2011) [9].

3 The $L_M{}^+$ algorithm and its application for inferring protocol state machines

We use Mealy machines to represent the state machine of a protocol as they can be used to model the behaviour of protocols using requests and responses (which are quite typical in practice) in a simpler way than finite state machines or Moore machines. Mealy machines differ from simple finite state machines in that for every state transition that is triggered by an input, an output is defined. The set I of possible inputs is called the input alphabet and the set O of possible outputs is called the output alphabet.

Angluin described an algorithm in [2] that can be used to infer minimal finite state machines, and this algorithm can be adapted for inferring Mealy machines as well. In this work, we adopt the techniques of Shahbaz and Groz described in [20], and from this point forward, we refer to the Mealy machine inferring algorithm described in [20] as $L_M{}^+$.

Since we use the $L_M{}^+$ algorithm as a black box, a high-level overview of its operation is sufficient for our purposes here. The $L_M{}^+$ algorithm is executed by a *learner*, and it requires a *teacher*. The teacher knows the Mealy machine to be inferred, and the task of the learner is to infer that machine. The teacher can answer two types of queries for the learner: first, for a certain sequence of input characters, the teacher returns the output of the machine to be inferred (input query); second, the teacher can determine whether a certain Mealy machine conjectured by the learner is the same as the one to be inferred (equivalence query). If the conjectured machine differs from the real one, then the teacher returns a counterexample: a sequence of input characters for which the real and the conjectured machines produce a different output.

A Mealy machine can be used to model the state machine of a client-server protocol in a fairly straightforward manner: the input alphabet of the machine contains the possible messages that the client may send to the server (i.e., the requests) and the output alphabet contains the possible messages that the server may send to the client (i.e., responses, acknowledgements, errors, *etc.*).

Clearly, including all possible individual messages in the input and output alphabets can easily lead to problems: a huge resulting Mealy machine and a very long running time of the $L_M{}^+$ algorithm. For instance, if a message contains a 4-byte timestamp, then the alphabet would contain at least 2^{32} elements to represent all possible messages containing different timestamp values. To bring the size of the alphabets to a manageable range, we represent message types by the elements

of the input and output alphabets instead of individual messages. A message type models a group of messages that have the same format but may differ in the specific values in the fields of the given message type.

In order to work with message types, we use two helper functions: a message classifier and a message generator. The message classifier function takes a particular message as input and returns its message type. The message generator function takes a message type as input and generates a valid message that has the specified message type. While the message classifier function should be deterministic, the message generator can be non-deterministic: the values of the message fields can be randomly generated as long as the message remains well-formed (i.e., consistent with its type). An additional function, a message updater is also useful, which takes as input the set M of all previously sent messages and a particular message m of this set, and returns a new message m' of the same type as m, such that the values of certain fields in m' are the mutations of the corresponding values in m, and M does not include m' yet. The updater function takes the semantics of certain fields into account: for instance, constant fields and identifiers are not mutated, a counter is mutated by incrementing it, *etc.* Such an awareness of semantics improves the efficiency and accuracy of our algorithm.

Recall that we aim at reverse engineering the protocol state machine of a system under test (SUT). To achieve this goal, we use the ${L_M}^+$ algorithm as the learner that infers the unknown Mealy machine representing the protocol, and we need to provide the teacher that answers the queries of the learner. We construct the teacher by using the helper functions defined above, and by sending messages to and observing the responses of the actual implementation of the protocol provided by the SUT.

This works in the following way: we start by using the message generator helper function to produce messages for every message type. We use these pre-generated messages to avoid a deterministic protocol appearing to be non-deterministic due to freshly generated random values used in the same message at different stages of our algorithm. Then, we run the ${L_M}^+$ learner and we respond to its queries. Input queries are answered by first resetting the SUT, then sending the pre-generated messages (after running the message updater helper function on them) corresponding to the learner's input query to the SUT, and finally running the message classifier helper function on the SUT's responses to get the message types that the learner can understand. Equivalence queries are answered by generating random input queries, running them against both the SUT and the conjectured machine, and comparing their outputs. The number of queries needed to decide about equivalence with a given confidence level has been studied in [2], and we follow those guidelines. Once the ${L_M}^+$ learner conjectures a Mealy machine that is deemed correct, our algorithm terminates with that machine as the output.

4 Extending the $L_M{}^+$ algorithm

The above-described version of our algorithm may return an incomplete protocol state machine because only a single message of each type is generated, which always triggers the same type of response, while it may be possible that other variants of the same message would result in a different response. Consider, for example, a request for reading the content of some memory address; the response can be the data found at the specified address or an error if the address was invalid. The extended version of our algorithm attempts to find these differing behaviours by finding messages of the same type that trigger different responses from the SUT. This further divides the set of messages that belong to a particular message type. We will call these new sets message subtypes. Our goal is to find as many subtypes as possible, ideally all of them. This is done by generating multiple messages of each type for the input alphabet I, then running the simple version of our algorithm with them. The resulting Mealy machine is analyzed, and messages that have the same type but do not produce different behaviour are removed (we call this step *deduplication*). More precisely, we say that two messages do not produce different behaviour, if, for all states of the Mealy machine, the two inputs result in the same output and the same new state. This means that deduplication only leaves a single message instance from every message subtype. After deduplication, new messages are generated and added to the set of possible inputs, and the simple algorithm is executed again. This is repeated again and again to reach more and more complete representations of the protocol as more subtypes are discovered (see Algorithm 1). A straightforward criterion for stopping could be requiring a number of runs where no new subtypes are found.

Initialization: $I := \emptyset$
Do
  $I := I \cup generateNewMessages()$;
  $M_{out} := L_M^+(I)$;
  $I, M_{out} := deduplicate(I, M_{out})$;
While *stopping criterion is not met*;
Result: M_{out}

Algorithm 1: Pseudocode of the extended algorithm

This method keeps the advantage of reducing the size of the input alphabet while still allowing us to discover inputs that are not considered different by message type categorization but reveal different behaviour of the protocol. In addition, not having an absolutely perfect message format becomes less of a problem, since message types that are not correctly separated by the used message formats will be automatically separated into message subtypes. However, it has two possible major pitfalls. The first problem arises if specific message subtypes only reveal their different properties if they are used in combination. As an example, let us

look at a message type that changes some kind of operating mode. Let there be three modes: A, B and C, where there is a subtype for changing into each of these modes (a, b and c respectively). All other message instances that belong to the mode changing message type do not have any effect. Starting out, the protocol is always in mode A, and each mode can only be changed into from a neighbouring mode (A is only accessible from B, B is accessible from both A and C, while C is only accessible if already in mode B). Now, suppose that only a and c is in the input alphabet along with some other messages from the same message type (but not b). In that case, c might be falsely discarded since it does not do anything unless the protocol is in mode B, but there is no way to make that happen with this collection of messages. This might be countered by increasing the number of message instances that are added each time, but this increases the queries needed to run the algorithm.

The second shortcoming is related to the generator function. As long as the generator produces message instances at a uniform distribution from each subtype, there is no problem; however, it is not trivial to write such a generator. For example, if a generator is generating read messages that read from a specific memory address, and fills the target address at random, the rate at which it will produce "valid" messages that result in a response with the read value and "invalid" messages that return an error because of an out of bound read depends on the memory layout of the device that is being tested. This could lead to never discovering one of the subtypes, or extending the runtime of the algorithm by selecting from only one subtype for many rounds and only later finding the other.

To deal with these problems, we need to have access to previously recorded traffic of the protocol that is being reversed – with information regarding resets. It is not unrealistic to have access to such traffic since network captures are usually used to reproduce the message types and semantics that are direct inputs to our algorithm. Of course, the quality and diversity of this capture will directly affect the results of the algorithm. We use the capture to find message instances that do not belong to any subtypes we have previously found. To do so, after each round, we compare the recorded communication with the predictions that the Mealy machine of the last round would make from the inputs. We classify the messages in the recorded traffic with the classifier, creating a sequence of message types, keeping track of which were queries (from client to server) and which were responses (from server to client). We feed the queries into a Mealy machine produced from the output of the last round and compare the outputs with the responses. If there is a difference, we know that a message in the communication up to that point is not covered in the Mealy machine, so all the queries in the flow are added to the input alphabet, and a new round is started. The modification is necessary because the Mealy machines produced by the ${L_M}^+$ algorithm in this setting expect message subtypes as inputs, but the message classifier can only produce message types.

To be precise, we actually convert the output Mealy machine into a simple Deterministic Finite Automaton (DFA) that takes inputs that are pairs of query and response messages. The DFA accepts a sequence if it could be produced by the Mealy machine. As a first step, we take the original machine and construct a

machine that is the exact copy of the Mealy machine, except each transition in the new Finite State Machine (FSM) is the pair of input and output message types in the original Mealy machine (the subtype is stripped here and duplicate transitions are discarded). Every state of this new machine is an accept state since every transition that is present here is represented in the original Mealy machine. We analyze the resulting machine to collect its alphabet. If the machine is incomplete (i.e., there are states where there is no transition with all letters of the alphabet, meaning there are some cases where the behaviour of the machine is not specified), we fully specify it by inserting the missing letters as transitions to the *FAIL* state. This is a new state that only has loop transitions, and is not an accept state. At each state, a missing input-output pair means that the original Mealy machine could not produce the given output for the given input in that state, so if this is encountered, the sequence is not covered. While analyzing a message sequence the first time the machine enters the *FAIL* state, the sequence up to that message pair can be used as new message instances for the next round. The same is true if we encounter a query/response pair that is not in the alphabet of the machine.

Now we have a machine that can handle the input we have, however, since we stripped away the subtypes, it may be non-deterministic (there may be multiple transitions from the same state with the same letter). This is not a problem, as there are known algorithms to transform non-deterministic state machines into deterministic ones. However, the resulting DFA may be exponentially large, which may be a problem for big inputs. It may be useful to run a minimization algorithm after determinization to get a minimal DFA.

One more improvement can be made by not completely stripping away the subtype information in the first step, but rather adding a transition with only the message type and also keeping the one with the subtype specified. The rest of the steps of creating the DFA remain the same. This is useful because for some of the messages, we can determine their subtype: these are the message instances that were used in the last round of learning. If there is an exact match with one of these, we can leverage this additional information.

5 Implementation and evaluation

We implemented the ${L_M}^+$ algorithm and our algorithm in Python using the NetworkX[1] package to represent Mealy machines. We designed the implementation to be modular, such that the different components are well separated and easy to replace. This is important for future improvements and to be able to easily plug in the functions that are different for each protocol (e.g., the message generator, the message classifier, and the part of the teacher that handles communication with the SUT).

[1] `https://networkx.github.io/`

5.1 Evaluation on a simple test protocol

For test and illustration purposes, we constructed and used a simple protocol which is illustrated in Figure 1. The protocol has 3 states: the starting state *DISCONNECTED*, the state *BASE*, where only *get* is a valid input, and the state *WRITE*, where both *get* and *write* are valid inputs. The difference between *get* and *bad_get* is that the parameter (target address) of the *get* message is valid, while it is invalid in a *bad_get*, and likewise with *write* and *bad_write*. Messages *get* and *bad_get* are of the same type, and similarly, messages *write* and *bad_write* have the same type. These two message types are used by the message generator to generate messages with random addresses; no recorded traffic analysis was used in this test.

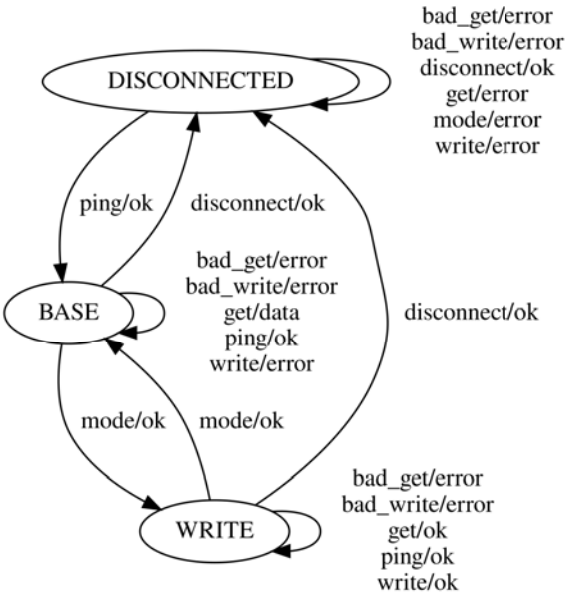

Figure 1: The Mealy machine of the simple test protocol

Figures 2, 3, and 4 show the inferred Mealy machine in different rounds after deduplication. The request messages are postfixed with a number; this is a counter showing how many times the message generator was called when the given message was generated. The starting state is *, and every other state is labelled by the messages that can be used in sequence to reach that state. In the first round, the algorithm generated two instances of each message type; however, each instance of the same message type produced the same behaviour, therefore, only one of them was kept (see Figure 2). In the second run, a new instance was generated from each message type, but these did not show new behaviour either, so they were discarded too. In the third round, a variant of the *write* message type was generated that resulted in an *ok* response, as opposed to the *error* response triggered by the

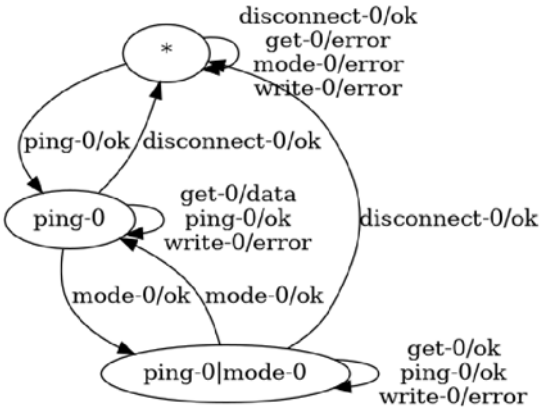

Figure 2: Output of the first round

other variant of the *write* message, so this was kept (see Figure 3). Finally, in the fourth round, the algorithm also finds a *get* message variant that results in different response observed so far, hence it is retained (see Figure 4). After five rounds with no new behaviour found, the algorithm stopped. As we can see in Figure 4, in our example, the Mealy machine inferred is identical to the state machine of the example protocol (except for the names of the states and message variants, of course).

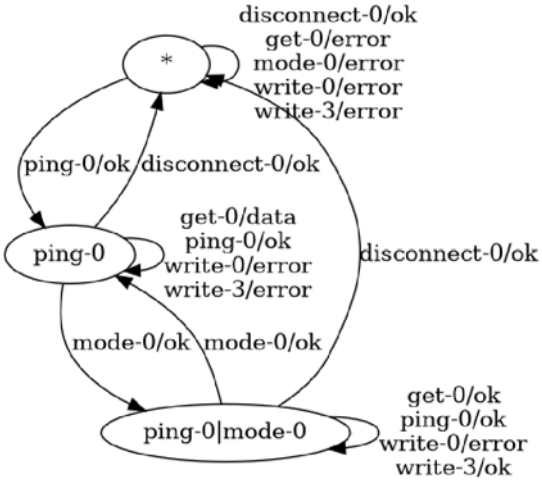

Figure 3: Output of the third round

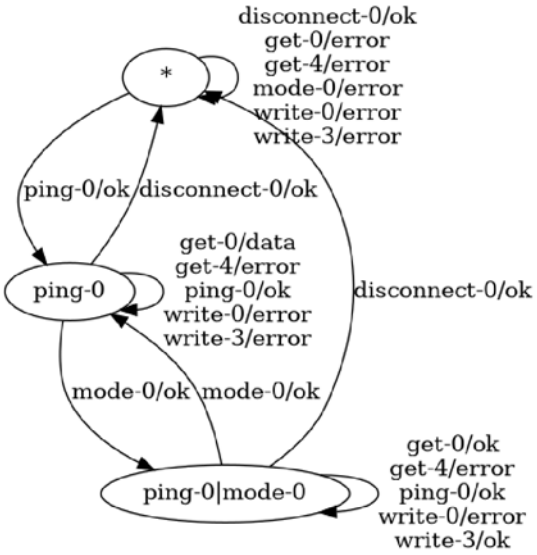

Figure 4: Output of the fourth round

5.2 Evaluation on Modbus

Modbus is a widely used industrial protocol that allows a client to read data from and write data to the server, usually a Programmable Logic Controller (PLC). For our tests, we used a custom implementation for both the client and the server.

Modbus messages begin with a fixed 8-byte header that is followed by a data part of variable length. For the classifier, we only used the last byte of the header as this byte determines the packet type. We constructed the generation function by classifying messages in the recorded network traffic and selecting from them when necessary, using the strategy described in Section 4.

In order to bridge the gap between real TCP communication and the abstract message instances described until now, some additional *artificial* message types were defined in the proxy that translates between the two. The first one is necessary because, to some queries, the server might simply not reply. When a certain timeout is reached without any response from the server, we conclude that it will not respond at all and return a *noresp* message as the response. The other possibility is the server terminating the connection because of some protocol violation, or because of a *DISCONNECT* packet. To model the communication properly, the proxy starts out with no open socket, and as long as there is no socket, it returns *noresp* to all queries. We define the *sockconn* query message, which results in the proxy opening a new TCP connection to the server and storing the socket – replacing the previously stored socket, if there was one –, and returning *sockconn*.

The output of our algorithm for Modbus can be seen in Figure 5. The input

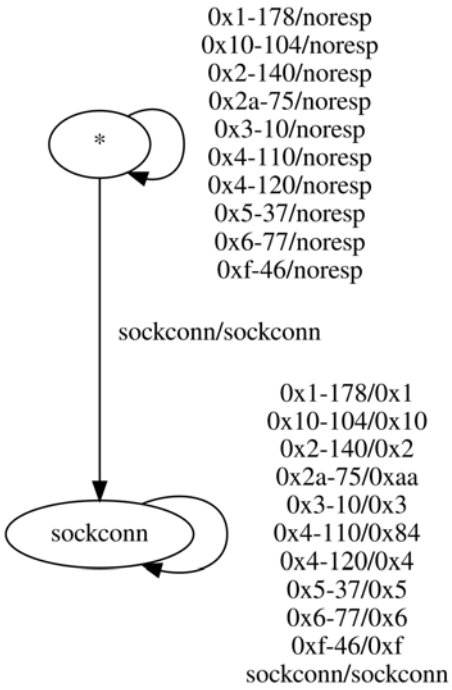

Figure 5: Resulting Mealy machine for Modbus

letters on the Mealy machine are the various function codes of Modbus, postfixed with a unique identifier to separate message subtypes. The output letters are the function codes that the server replied with. The figure shows that from a protocol state machine point of view, the Modbus protocol is very simple as it has two states: in the starting state only the *sockconn* message results in any response, as there is no connection to send or receive the other messages, and in the other state, the server always responds to queries. For each request (function code), the response has the same code as the query, except if an error is detected, in which case the response code is the query code plus $0x80$.

In our case, for request code $0x2a$, we always get sent back an error because it is an invalid function code, but for $0x4$ there is a subtype for a correct version of the query and one for the incorrect version. The network capture we used did not contain invalid uses of the other function codes, so those are not present in the produced Mealy machine.

Based on the open specification of the protocol [18], these results are 100% accurate.

5.3 Evaluation on MQTT

MQTT is a messaging protocol implementing a publish-subscribe model. Clients can subscribe to channels and will receive all messages posted to these channels. This protocol is ideal for evaluation because its simplicity and open specification allow easy verification. There are also plenty of free implementations for servers and client libraries, making it convenient to set up a test environment. For the client-side, we used the Eclipse Paho™ MQTT Python Client[2] library, and for the server-side we used Eclipse Mosquitto[3].

MQTT messages are comprised of a fixed header, a variable header and a payload. We only used the first four bits of the fixed header for the classifier as these bits determine the packet type according to the documentation. The second byte was used as well, to determine the length of the message, in case more than one message was sent in a single IP packet. For the classifier and the generation functions, we have taken the same approach as in the case of Modbus, with the exception that a third artificial message that is specific to MQTT was also defined. This message, called *pubtrigger*, simulates other clients connected to the MQTT server, and triggers a publish to one of the channels.

The MQTT protocol has three different Quality of Service (QoS) levels that govern how messages are published. As the level of QoS increases, so do the number of roundtrips and the confidence of the message reaching its destination. We used only messages with QoS 0, and in separate tests, we generated traffic that used one, two and three channels for posting messages. We configured the MQTT server with two different username and password combinations and forbade anonymous connections. Then we performed randomized actions through the client library by selecting from connecting, publishing, subscribing, unsubscribing, and disconnecting from the server. The channel and the message, where applicable, were also randomly selected. The generated network traffic was captured and used for the construction of the generation function and to prioritize the selection of message instances. As it can be seen on final results of the algorithm (Figures 6 and 7), at this level of QoS, the protocol is relatively simple: one can subscribe and publish to any channel after connecting with the correct credentials.

The outputs produced by the algorithm are correct based on the MQTT protocol specifications [4], taking into account that we only used a subset of the possible functions of the protocol.

We can see that membership in the various channels is entirely independent. As the number of potential channels increases, so do the number of states needed to keep track of how many of them the client is currently subscribed to. One state is necessary for each combination of channels that are joined, meaning two states when only one channel is used (the client is either subscribed or not), four with two channels (subscribed to none, either one, or both channels), and eight if there are three channels. Generally, the formula is 2^n, where n is the number of channels that are used. After reaching a high enough n, this is not very useful as it makes the

[2] `https://github.com/eclipse/paho.mqtt.python`
[3] `https://github.com/eclipse/mosquitto`

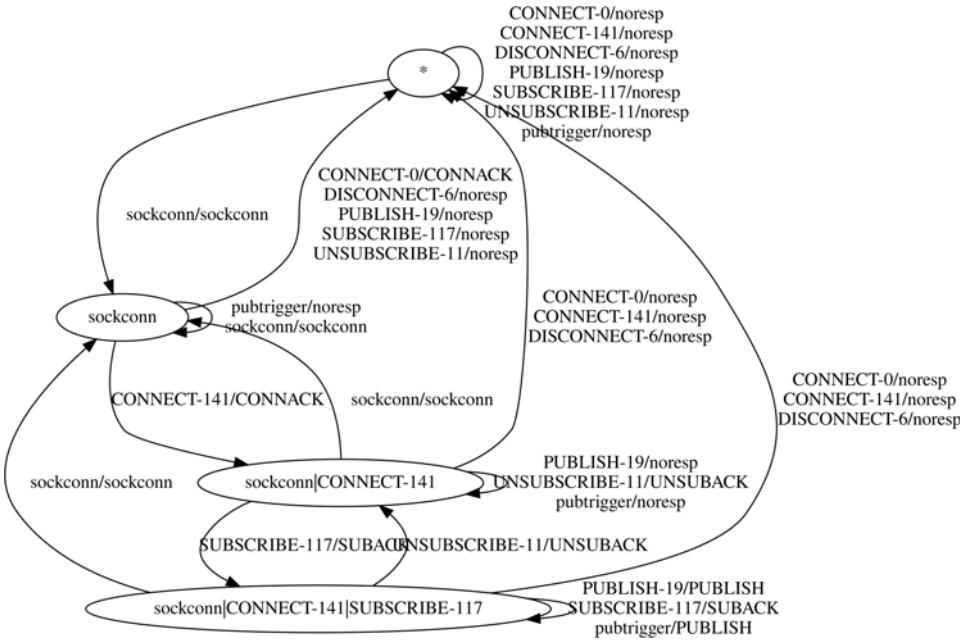

Figure 6: Resulting Mealy machine for MQTT with a single channel and QoS 0

analysis of the Mealy machine difficult because of its size, and it does not provide any new information. This is a limitation of using Mealy machines to represent the protocol. To counteract this problem, the number of such possible combinations should be kept to a low number. In our case, this means keeping the number of possible channels low.

We came up with a method that may help find such independent message subtypes. First, we define a *projection* of a Mealy machine on a set S of input letters. For all input letters in the input alphabet that are not in S, we ignore the output letter. In the practical implementation, we modify the output on the transitions that contain these letters, we change the output letter to a fixed value, in our case *noout*. After this, we run a minimization algorithm on the new Mealy machine, and the result will be the projected machine. During the minimization, all input letters that are not in S and do not influence the outputs of the transitions that contain letters that are in S completely disappear. At first, we project to single letters of the input alphabet, and then for each of these projections, we try to add another input letter to S in addition to the original one. If the resulting Mealy machine has the same number of states as the previous projection, we keep the letter in S; otherwise we remove it. After repeating with all input letters, we can take all the projections with a different set S and analyze them. These Mealy machines can be used together to replace the original Mealy machine, giving the inputs to all of

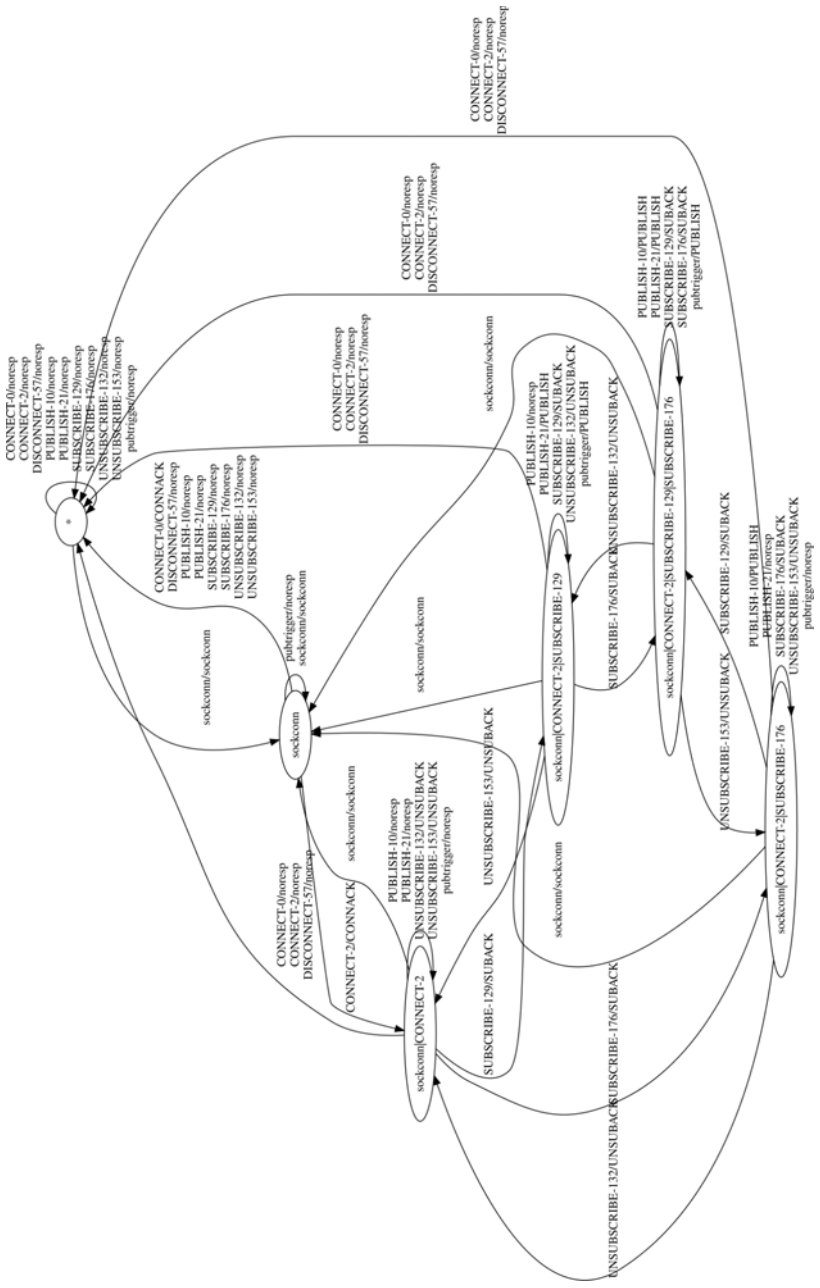

Figure 7: Resulting Mealy machine for MQTT with two channels and QoS 0

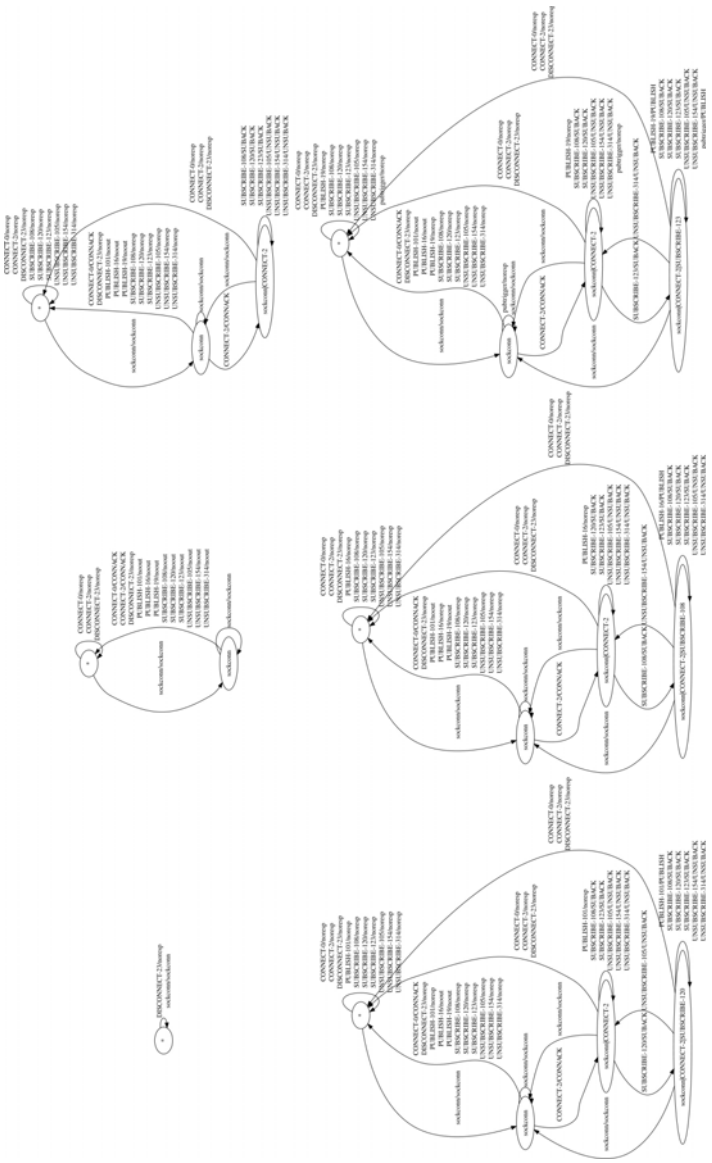

Figure 8: Results of the projection generation algorithm on the Mealy machine of MQTT with three channels with various sets of S. Each of the last three machines corresponds to a single channel. The figure is intended to illustrate the complexity of the resulting state machines only, the reader is not expected to be able to read the labels of the states or transitions.

them, and taking the output from either one which can produce it (if either Mealy machine does not have a transition with the particular letter at its current state, it should just do nothing). For the projections produced in this manner for QoS level 0 and three channels, the Mealy machine can be seen in Figure 8. The three machines in the bottom row can be grouped as implementing the same logic, only with different subtypes, and indeed, each corresponds to a single channel.

When testing with higher QoS, the same problems come up as with using multiple channels, but this time around the state of publishing is the culprit behind the state explosion. The above-described technique did not work so well in separating any independent parts of the machine.

6 Conclusion

In this work, we build on Shahbaz and Groz's L_M^+ algorithm (an adaptation of Angluin's L^* algorithm) and make use of captured network traffic as well as live queries to a known good protocol implementation in order to infer the state machine of an unknown protocol. We assume that message types and semantics have already been reverse engineered, and we use this information to intelligently and efficiently choose the possible inputs to use when querying the system under test. We demonstrate our methods on an example protocol created for this purpose, in addition to two real-world protocols, Modbus and MQTT. We show that our approach works by comparing the resulting automata to the original specifications.

In the future, we plan to investigate more complicated Mealy machine decomposition algorithms to produce hierarchical Mealy machines that can represent the protocol state machine more concisely. In addition, we aim to measure and further optimize input selection strategies.

References

[1] Andrew, Tridgell. How samba was written. `https://download.samba.org/pub/tridge/misc/french_cafe.txt`, 2003.

[2] Angluin, Dana. Learning regular sets from queries and counterexamples. *Information and Computation*, 75(2):87–106, nov 1987. DOI: `10.1016/0890-5401(87)90052-6`.

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

[4] Banks, Andrew and Gupta, Rahul. MQTT Version 3.1.1 Plus Errata 01 (OASIS Standard Incorporating Approved Errata 01). `http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/mqtt-v3.1.1.html`, 2015.

[5] Beddoe, Marshall A. Network protocol analysis using bioinformatics algorithms, 2004. http://www.4tphi.net/~awalters/PI/PI.html.

[6] Bermudez, Ignacio, Tongaonkar, Alok, Iliofotou, Marios, Mellia, Marco, and Munafò, Maurizio M. Towards automatic protocol field inference. *Computer Communications*, 84:40–51, 2016. DOI: 10.1016/j.comcom.2016.02.015.

[7] Bossert, Georges, Guihéry, Frédéric, and Hiet, Guillaume. Towards automated protocol reverse engineering using semantic information. *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security*, pages 51–62, 2014. DOI: 10.1145/2590296.2590346.

[8] Cho, Chia Yuan, Babi ć, Domagoj, Shin, Eui Chul Richard, and Song, Dawn. Inference and analysis of formal models of botnet command and control protocols. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS '10, page 426–439, New York, NY, USA, 2010. Association for Computing Machinery. DOI: 10.1145/1866307.1866355.

[9] Cho, Chia Yuan, Babić, Domagoj, Poosankam, Pongsin, Chen, Kevin Zhijie, Wu, Edward XueJun, and Song, Dawn. MACE: Model-inference-assisted concolic exploration for protocol and vulnerability discovery. *SEC'11 Proceedings of the 20th USENIX Conference on Security*, pages 139–155, 2011.

[10] Comparetti, Paolo Milani, Wondracek, Gilbert, Kruegel, Christopher, and Kirda, Engin. Prospex: Protocol specification extraction. *30th IEEE Symposium on Security and Privacy*, pages 110–125, 2009. DOI: 10.1109/SP.2009.14.

[11] Cui, Weidong, Kannan, Jayanthkumar, and Wang, Helen J. Discoverer: Automatic protocol reverse engineering from network traces. *SS'07 Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium*, 2007. https://www.usenix.org/conference/16th-usenix-security-symposium/discoverer-automatic-protocol-reverse-engineering-network.

[12] Cui, Weidong, Paxson, Vern, Weaver, Nicholas C., and Katz, Y H. Protocol-independent adaptive replay of application dialog. In *Network and Distributed System Security Symposium*, 2006. https://www.ndss-symposium.org/ndss2006/protocol-independent-adaptive-replay-application-dialog/.

[13] Duchêne, Julien, Guernic, Colas Le, Alata, Eric, Nicomette, Vincent, and Kaâniche, Mohamed. State of the art of network protocol reverse engineering tools. *Journal of Computer Virology and Hacking Techniques*, 14(1):53–68, 2017. DOI: 10.1007/s11416-016-0289-8.

[14] Ládi, Gergő, Buttyán, Levente, and Holczer, Tamás. 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. DOI: 10.36244/ICJ.2020.2.4.

[15] Lee, David and Yannakakis, Mihalis. Principles and methods of testing finite state machines – A survey. *Proceedings of the IEEE*, 84(8):1090–1123, 1996. DOI: 10.1109/5.533956.

[16] Leita, C., Mermoud, K., and Dacier, M. ScriptGen: an automated script generation tool for Honeyd. *21st Annual Computer Security Applications Conference*, pages 203–214, 2005. DOI: 10.1109/CSAC.2005.49.

[17] Luo, Jian-Zhen and Yu, Shun-Zheng. Position-based automatic reverse engineering of network protocols. *Journal of Network and Computer Applications*, 36(3):1070–1077, 2013. DOI: 10.1016/j.jnca.2013.01.013.

[18] Modbus Organization, Inc. Modbus application protocol specification v1.1b3. http://www.modbus.org/docs/Modbus_Application_Protocol_V1_1b3.pdf, 2012.

[19] Narayan, John, Shukla, Sandeep K., and Clancy, T. Charles. A survey of automatic protocol reverse engineering tools. *ACM Computing Surveys*, 48(3), 2016. DOI: 10.1145/2840724.

[20] Shahbaz, Muzammil and Groz, Roland. Inferring mealy machines. In *International Symposium on Formal Methods*, pages 207–222. Springer, 2009. DOI: 10.1007/978-3-642-05089-3_14.

[21] Wang, Y., Xiaochun Yun, Shafiq, M. Z., Wang, L., Liu, A. X., Zhang, Z., Yao, D., Zhang, Y., and Guo, L. A semantics aware approach to automated reverse engineering unknown protocols. *20th IEEE International Conference on Network Protocols (ICNP)*, pages 1–10, 2012. DOI: 10.1109/ICNP.2012.6459963.

[22] Wang, Yipeng, Li, Xingjian, Meng, Jiao, Zhao, Yong, Zhang, Zhibin, and Guo, Li. Biprominer: Automatic mining of binary protocol features. *12th International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT)*, pages 179–184, 2011. DOI: 10.1109/PDCAT.2011.25.

[23] Wang, Yipeng, Zhang, Zhibin, Yao, Danfeng, Qu, Buyun, and Guo, Li. Inferring protocol state machine from network traces: A probabilistic approach. *ACNS 2011: Applied Cryptography and Network Security*, pages 1–18, 2011. DOI: 10.1007/978-3-642-21554-4_1.