Towards Reverse Engineering Protocol State Machines

Gábor Székely, Gergő Ládi, Tamás Holczer and Levente Buttyán

Abstract: In this work, we are addressing the problem of inferring the state machine of an unknown protocol. Our method is based on prior work on inferring Mealy machines. We require access to and interaction with a system that runs the unknown protocol, and we serve a state-of-the-art Mealy machine inference algorithm with appropriate input obtained from the system at hand. We implemented our method and illustrate its operation on a simple example protocol.

Keywords: Automated Protocol Reverse Engineering, State Machines, Mealy Machines

Introduction

Many systems use closed protocols whose specification is not made publicly available. Examples include industrial control systems and in-vehicle embedded networks. Often, it would be very beneficial to understand those closed protocols. For instance, network anomaly detection tools cannot monitor industrial control systems without understanding their protocols, hence, cannot detect potential cyber attacks on them.

Protocol reverse engineering is the activity of uncovering the specification of an unknown protocol. This can be a tedious work, so automation is required to make it practical. The goal of automated protocol reverse engineering methods can be two-fold: determining the format of messages used by the protocol and recovering the state machine of the protocol. In another paper [2], we studied the problem of determining the message formats of unknown binary protocols, and developed a tool which can take captured network traffic containing messages of the protocol and output the identified message types and the semantics of message fields for the different message types. In this work, we are addressing the problem of inferring the state machine of an unknown protocol, and we assume that message types have already been identified (e.g., by using our tool mentioned above).

Our method is based on prior work on inferring Mealy machines, and in particular, on the work of Shahbaz and Groz [3]. We use their Mealy machine inference algorithm and extend it with elements that make it possible to use their conceptual results in practice to reverse engineer the state machine of real-world protocols in an automated way. Our method requires access to and interaction with a system that runs the unknown protocol, and it basically consists in serving the Mealy machine inference algorithm of Shahbaz and Groz with appropriate input obtained from the system at hand.

Inferring Mealy Machines

We use Mealy machines to represent the state machine of a protocol, as they can be used in a more straightforward manner to model the behavior of protocols using requests and responses (which is quite typical in practice) than finite state machines or Moore machines can. 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 [1] that can be used to infer minimal finite state machines, and this algorithm can be adapted for inferring Mealy machines too. In this work, we adopt the techniques of Shahbaz and Groz described in [3], and in the sequel, we refer to the Mealy machine inferring algorithm described in [3] 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.

Applying and extending the $L_M{}^+$ algorithm

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 can contain the possible messages that the client may send to the server (i.e., the requests) and the output alphabet can contain 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 in 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 that may differ in the specific values in the fields of the given message type.

In order to work with messages 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 into account the semantics of certain fields: 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 above defined helper functions, and by sending messages to and observing 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 [1] 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.

However, the above described version of our algorithm may return an incomplete protocol state machine, because it may happen that only a single message of a given 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 additional behaviors by finding messages of the same type that trigger different responses of the SUT. This is done by generating multiple messages of each type of the input alphabet I, and 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 behavior are removed (we call this step *deduplication*). Then, new messages are generated and added to the set of possible inputs, and the simple algorithm is executed again. This is repeated until a certain amount of runs in a row do not generate new messages that induce different behavior.

Implementation and evaluation on a simple protocol

We implemented the ${L_M}^+$ algorithm and our algorithm in Python, using the NetworkX[1] package for representing 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 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).

For testing 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.

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 labeled by the messages that can be used in sequence to reach that state. In the first round, the algorithm generates two instances of each message type, however, each instance of the same message type produced the same behavior, therefore, only one of them were kept (see Figure 2). In the second run, a new instance was generated from each message type, but these did not show new behavior 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 other variant of the *write* message, so this was

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

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 5 rounds with no new behavior 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).

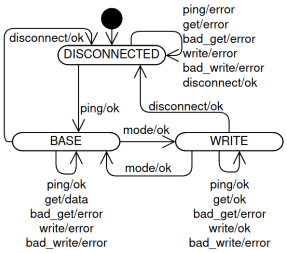

Figure 1: Protocol Mealy machine

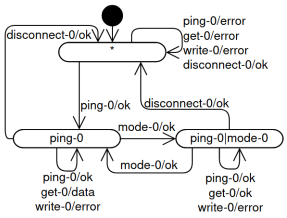

Figure 2: First round output

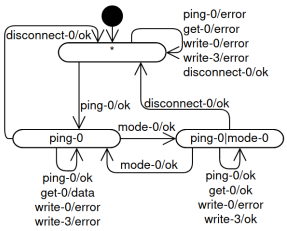

Figure 3: Third round output

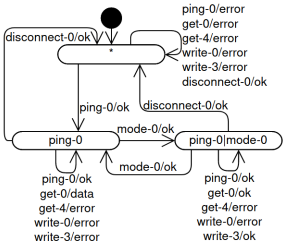

Figure 4: Fourth round output

Acknowledgment

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).

References

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

[2] Gergő Ládi, Levente Buttyán, and Tamás Holczer. Message format and field semantics inference for binary protocols using recorded network traffic. In *IEEE Conference on Software, Telecommunications and Computer Networks (SoftCom)*, September 2018.

[3] Muzammil Shahbaz and Roland Groz. Inferring mealy machines. In *International Symposium on Formal Methods*, pages 207–222. Springer, 2009.