On the Performance Evaluation of Protocol State Machine Reverse Engineering Methods

Gergő Ládi, and Tamás Holczer

***Abstract*—Having access to the specifications of network protocols is essential for several reasons in IT security. When the specifications are not known, one may turn to protocol reverse engineering methods to reconstruct these, typically by analysing recorded network traffic or inspecting an executable that implements the protocol. First, the format and structure of the messages need to be recovered, then the state machine of the protocol itself. Over the years, several solutions have been proposed for both tasks. As a consequence, picking the right solution for a given scenario is often a complex problem that involves evaluating and comparing various solutions. In this paper, we review the current means of evaluating the performance of protocol state machine reverse engineering methods. To help alleviate the shortcomings of the current methodology, we propose two new metrics of performance to be measured: correctness and completeness of output for partial runs (when runtime is bounded). These, combined with previously used metrics should make it easier to pick the most ideal choice for a given use case. We also propose the examination of cases where the algorithms have to work with incomplete or inaccurate syntactical information. We showcase how these new metrics and related information may be useful for the evaluation and comparison of various algorithms by applying these new methods to evaluate the performance of a recent protocol state machine reverse engineering method.**

***Index Terms*—protocol reverse engineering, protocol state machine, performance evaluation, runtime analysis, bounded runtime, incomplete input.**

I. Introduction

Communication protocols describe the formats, contents, and sequences of messages that are sent and received in order to exchange instructions and data between the communicating parties, along with the rules according to which these messages need to be processed. Knowing this information is essential for various use cases in the domain of IT security. For example, knowing the protocol enables fuzz testing software that implements the protocol for programming errors or hidden features such as backdoors [1]. It also makes it possible to develop intrusion detection/prevention systems that understand the protocol and can trigger alarms or even block the source when anomalous or malicious protocol messages are detected on the network [2]. One may also build honeypots and honeynets that simulate a device or a network of devices speaking a protocol in order to attract attackers and analyse their behaviour [3]. Unfortunately, the specifications in which these protocols are defined are not always publicly available.

By applying protocol reverse engineering methods, it becomes possible to reconstruct the message formats and syntax, as well as the state machine of the protocol. While it is possible to perform protocol reverse engineering manually, manual methods are considered time-consuming and error-prone. Given that results may be needed as quickly as possible and that new protocols appear frequently, automated approaches are preferred. However, the two may also be combined – results from an automated approach may also serve as a basis for later manual analysis. Automated solutions generally draw conclusions based on recorded network captures, generate network traffic and analyse the responses, inspect a binary (executable) that contains an implementation of the protocol to be reverse engineered, or a combination thereof [4]. It has been shown [4] that approaches based on binary analysis can achieve better results, however, legal agreements may prohibit such reverse engineering, and binaries may not always be available.

Protocols may be classified as binary or plain text, based on how their messages are represented. Binary protocols, such as Modbus, Message Queuing Telemetry Transport (MQTT), or Border Gateway Protocol (BGP) exchange binary messages that are not human-readable. In these messages, the meaning of each byte at a given position (which may be fixed or calculable) is known in advance, there are no field separators, and generally, there is a byte (or group of bytes) that specifies which of the possible message types it belongs to. Plain text protocols, such as File Transfer Protocol (FTP) or Internet Message Access Protocol (IMAP), on the other hand, exchange human-readable messages consisting of fields that are separated by delimiters (e.g. spaces, tabs, or line feed characters). Here, one of the fields contains a command or keyword that determines which of the possible message types it belongs to.

Regardless of the above classification, the process of reverse engineering a protocol consists of three major phases [5]. The first phase is the preparatory phase, in which the environment for the analysis is set up. If using an approach that relies on recorded network traces, then network traffic is generated and recorded in this phase as well. The second phase aims to discover the possible message types of the

Manuscript received November 20, 2023; revised January 4, 2024. Date of publication February 6, 2024. Date of current version February 6, 2024. The associate editor prof. Miljenko Mikuc has been coordinating the review of this manuscript and approved it for publication.

Authors are with the Laboratory of Cryptography and System Security, Department of Networked Systems and Services, Budapest University of Technology and Economics, Budapest, Hungary (e-mails: {gergo.ladi, holczer}@crysys.hu).

Digital Object Identifier (DOI): 10.24138/jcomss-2023-0149

protocol, along with the data type (e.g. string, integer, etc.) and meaning (semantics, e.g. *a temperature value*) of each byte or series of bytes in these messages. Sometimes also referred to as *commands* or *operations*, message types are unique structures of data that each convey a different meaning and need to be interpreted differently by the receiver. For example, a protocol might have a *read command* consisting of a four-byte integer specifying the address whose value should be read, followed by a two-byte integer specifying the number of bytes to read. Another message type could be a *write command* that also carries an address, the number of bytes to be written, as well as the values of the bytes that need to be written. Finally, in the third phase, the state machine of the protocol is reconstructed. This state machine dictates what action should be taken when a specific sequence of message types is received, starting from a known initial state. For example, a client connecting to a server might start from an *unauthenticated* state, where only *authentication command*s are accepted, but once successfully authenticated, *read* and *write* commands are also accepted.

This paper reviews the current means and methods of evaluating the performance of protocol state machine reverse engineering methods. Addressing the shortcomings of the current methodology, we propose two new metrics of performance: completeness and correctness of output with bounded runtime. We also propose the inclusion of the analysis of the effects of relying on incomplete or partially incorrect input.

The rest of the paper is organized as follows: in Section II, we discuss related work, briefly going over the history of protocol reverse engineering, culminating in the most recent advances. Next, in Section III, we enumerate the current aspects of performance evaluation: commonly used metrics of quality, and runtime and complexity. Also in Section III, we detail our newly proposed metrics and methods. Then, in Section IV, we show how the previously discussed aspects are useful and how they may be evaluated in practice: we first give a summary of a recent protocol state machine reverse engineering method by Székely et al. [6], then proceed by evaluating the method according to each aspect. Finally, Section V concludes our paper.

II. Related Work

The history of protocol reverse engineering dates back to the 1950s. At that time, it was generally used for fault analysis in electrical circuits that implemented a finite state machine that represented a protocol [7]. At around the turn of the century, computers and computerized accessories became increasingly common. With the Internet also becoming widespread, the number of network applications increased, which led to the proliferation of new network protocols. A number of these were undocumented, or at least had no publicly available documentation. This meant that in order to develop compatible applications, for example, an alternative server for a discontinued chat service, the protocol had to be reverse engineered.

The first major project that aimed at recovering the specifications of an unknown protocol was the Samba Project (2003) [8] that intended to reverse engineer the specifications of Microsoft's SMB (Server Message Block) protocol. The authors used a network trace based approach along with active random probing to identify message types and semantics. M. A. Beddoe's Protocol Informatics Project [9] was next in 2004, which employed bioinformatical algorithms on network traces to infer the message types of the text-based protocol HTTP. RolePlayer (2006) [10], Discoverer (2007) [11], Biprominer (2011) [12], ReverX (2011) [13], ProDecoder (2012) [14], and AutoReEngine (2013) [15] soon followed, all of which relied exclusively on network traffic.

The majority of these early works focused on inferring message types and formats only. Little emphasis was put on inferring field semantics, and none attempted to recover the state machine of the protocols. The first works that tried to infer field semantics did not achieve significant results – Discoverer achieved between 30 and 40% accuracy [11], and Netzob was still below 50% on average [16]. However, these results proved that recovering field semantics was not an impossible task. In 2015, FieldHunter [17] was published, reaching over 80% accuracy on semantics. In 2020, GrAMeFFSI [18] was shown to achieve over 90% accuracy when high-quality network captures are available, followed by IPART [19] in the same year that had results between 70 and 100%.

While some of these algorithms, such as FieldHunter, were designed to be able to reverse engineer both binary and text-based protocols, others specialized in either. Biprominer, as its name suggests, targeted binary protocols, as did GrAMeFFSI and IPART, while ReverX targeted text-based protocols only. The more specialized algorithms usually achieved higher performance scores compared to the more general solutions of their time. Their inner workings vary: RolePlayer and Discoverer rely on sequence alignment, Biprominer and AutoReEngine leverage data mining approaches, ProDecoder makes use of natural language processing algorithms, GrAMeFFSI employs graph analysis, while IPART makes use of a voting expert mechanism.

Before the state machine of a protocol can be reverse engineered, the message types and formats need to be known, as this information is not only used for the classification of protocol messages (i.e. determining which message type a specific message belongs to), but also for the generation of new messages (in the case of algorithms that generate and send new messages). Thus, these reverse engineering methods must either recover this information themselves or rely on existing message format inference methods. Based on recent surveys [5][20], so far, there have been fewer attempts to reconstruct protocol state machines than to recover message types and semantics. The majority of the solutions are based on network traces instead of binary analysis, and the majority of these trace based solutions are passive, meaning that only captured network traffic is used as input, no live systems running an implementation of the protocol are interacted with.

The first notable method for state machine inference was ScriptGen (2005) [21]. Like Beddoe's previously mentioned project, it uses the Needleman-Wunsch sequence alignment algorithm, 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. ScriptGen was later followed by Cho et al.'s work on reverse engineering the protocol of the MegaD botnet (2010) [22] that leveraged and optimized Angluin's L^* algorithm [23] to infer a Mealy machine based on network traces and live queries. They achieved between 96% and 99% accuracy on various protocols. Another important result was Veritas (2011) [24], which employs statistical analysis on captured network traffic to build probabilistic protocol state machines that are claimed to be 92% accurate on average. Yet another notable mention is PREUGI (2017) [25], which is based on error-correcting grammatical inference, and also achieves an accuracy that is over 90%. In 2021, Székely et al. [6] published a state machine inference method that is based on the ${L_M}^+$ Mealy machine inference algorithm of Shahbaz and Groz [26], which itself is an adaptation of Angluin's L^* algorithm for learning regular languages [23]. This method employs smart message generation to significantly reduce the number of messages that need to be sent, making it usable in practice. Despite the reduction in the number of messages sent, it achieved perfect scores on several test cases. There also exist approaches that rely on binary analysis instead of network traces, the most significant ones being Prospex (2009) [27] and MACE (2011) [28].

Protocol reverse engineering is a field that is still actively studied. New methods surface every year [20], some of them being reiterations of previous algorithms, some of them being completely new methods that are built on state-of-the-art applications such as machine learning. Some recent examples are an algorithm by Yang et al. [29] that leverages deep learning, and another one by Wang et al. [30] that employs convolutional neural networks. Probabilistic inference is also a novel approach that is applied by NetPlier [31], while an even more recent algorithm is PREIUD [32], which makes use of deep neural networks to infer message formats and semantics for industrial control protocols.

III. Performance Evaluation

With numerous approaches, methods, and algorithms having been published, one may be faced with quite a dilemma trying to determine the best option for a given use case. Some approaches expect an executable implementing the protocol to be reverse engineered, while others expect recorded network traffic to be available, or possibly the existence of a remote implementation that may be used as an oracle. Some methods work only with binary protocols, others only with text-based ones, while there also exist more general approaches that can work with both. Some offer quicker results at the cost of quality, while others take longer to run but provide higher-quality results. Some methods expect extra conditions to be satisfied, but may offer better results than more general ones.

In any case, it is essential to have a means of evaluating the performance of various algorithms so that they can be compared to one another. This comparison, however, is often an arduous task – authors employ various metrics to showcase the significance of their results, but these may be named, defined, and used differently by the different authors. A second issue that arises is that even when these metrics are the same (or can be calculated from one another), the data sets over which the algorithms are evaluated are seldom the same due to the lack of a standard data set. Testing with a common data set is also difficult because the implementations themselves are rarely published.

Nevertheless, this section aims to provide an overview of the current and newly proposed aspects of performance evaluation: commonly used metrics of quality, runtime and complexity, the effects of (partially) incorrect or incomplete input, and bounded runtime.

A. Common Quality Metrics

The metrics of quality intend to describe the quality of the output in various manners. The most commonly used ones are correctness, completeness, and conciseness (or variants thereof).

Correctness is a measure of how accurately the *found* elements of the inferred message type specifications or protocol state machine match the true specifications or state machine. Elements that were found but do not actually exist in the true specifications lower this score, while missing or duplicate elements do not affect it. It is generally expressed as a percentage between 0 and 100%. A correctness of 100% means that everything that was found is correct, but it does not guarantee that everything was found. For example, the state machine could be missing states and transitions. Correctness is similar to *precision* in classification problems.

Completeness (also referred to as coverage) measures what portion of the original message types and formats is present in the inferred specifications (or states and transitions in the case of state machines). It is also expressed as a percentage between 0 and 100%. A completeness of 100% means that everything was found, but it does not guarantee that everything that was found is correct. For example, the resulting state machine could contain extra states that were not in the original state machine. Completeness is similar to *recall* in classification problems.

Conciseness measures how verbose the result is, compared to the original, or, in other words, how many found elements represent one true element. A lower number is better. A conciseness of 1 means that there are no duplications, while a score of 1.5 means that half of the elements are represented twice. For example, detecting one single message type as two distinct message types increases the conciseness score. If these message types are later used for state machine reverse engineering, the resulting state machine will have duplicate states and transitions compared to the original one, also increasing this score.

B. Runtime, Time Complexity

Another typical question is the speed of the algorithm, i.e. how its runtime depends on the number and the properties of its inputs. Usually, three cases are considered: the worst case, the best case, and the average case.

Of these, the most relevant case is the worst case: how long the algorithm takes to finish when given inputs that result in the most possible computations being performed in the background. Worst-case complexity is the property by which the performance of algorithms is compared, and by which fitness for a particular purpose is determined. Generally, it cannot be expressed as an exact function of the (size of the) inputs, so approximations are used instead. Worst cases are approximated using an asymptotic upper bound, denoted by O (e.g. $O(n^2)$), also known as the big O notation.

Best case calculations, on the other hand, aim to determine the minimum amount of time the algorithm needs to terminate. Since these calculations include trivial and sometimes unrealistic inputs, best-case complexity is not meaningful in practice. In a similar fashion to worst cases, the runtime of the best case cannot typically be expressed as an exact function, and approximations are used instead. Best cases are approximated using an asymptotic lower bound, denoted by Ω (e.g. $\Omega(log(n))$), also known as the big Omega notation.

Finally, average case calculations are intended to show how long the algorithm takes to run to completion on average, with relation to (the size of) the inputs. Average cases are approximated using both an upper and a lower bound (also called a tight bound), denoted by Θ (e.g. $\Theta(n)$), also known as the big Theta notation.

C. Correctness and Completeness for Partial Runs

Prior analyses only focused on the previously discussed common quality metrics and the complexity of the algorithms. However, other metrics could exist to aid in the selection process, such as correctness and completeness for partial runs. There are scenarios where the number of requests made to the implementation or the window of time that is available for running the algorithm is limited, or having earlier access to a partial state machine is more important than having the complete automaton but only at a later point in time. For example, a malware analyst who is trying to understand the command and control (C&C) protocol of a new botnet might want to fly under the radar and limit the number of requests that are sent to the C&C server over given periods of time in order not to get detected. If it takes too long, the server might be rotated or taken offline while the analysis is running, and even if it stays up long enough to run the algorithm to completion, it would be beneficial for the analyst to be able to see a partial state machine while the rest of the algorithm is running. With a partial state machine, they can start documenting the protocol and perhaps develop a custom implementation of the botnet server that may be used to gain a better understanding of the clients.

The process of measuring correctness and completeness for partial runs is similar to that of measuring these metrics for a complete run, except there are limits concerning the time within which the algorithm must terminate or the number of queries the algorithm may issue to an actual implementation of a protocol. Depending on the inner logic of an algorithm, it may not be possible to stop the algorithm at arbitrary points (if at all) and retrieve a partial result, making it difficult (if not impossible) to measure these metrics effectively and accurately.

D. Working with Incorrect or Incomplete Information

Reviewing the state-of-the-art message format reverse engineering algorithms, it can be stated that while there exist algorithms that are capable of achieving a 100% score in completeness and correctness for certain protocols, there is no guarantee that these will always perform so well on all, including presently unknown protocols. In addition, if a network trace based algorithm is used, its success also largely depends on the diversity and number of messages captured in the network traces. Message types that have never been seen will not be included in the inferred specifications, and message types of which there was not enough in the traces (or these were not diverse enough) are likely to have inaccurately classified fields.

The shortcomings of inferred message formats may be grouped into three wider categories: missing message types, duplicate message types, and incorrectly inferred message types. Each of these may have different effects on the result of a protocol state machine inference algorithm. Possible effects include missing states and transitions from the inferred state machine, duplicate transitions, and even the algorithm not being able to produce an output (e.g. if an internal contradiction is reached). Effects may also differ based on whether the missing, duplicate, or incorrect message type is a client-to-server or a server-to-client message type.

Investigating how these cases are handled by the algorithm and what effects these have on the resulting state machine could be another factor to consider when evaluating, comparing, and choosing algorithms, especially when the input is expected to be imperfect.

IV. A Practical Example

This section is divided into four parts. First, in subsection IV-A, we summarize the algorithm to be evaluated so that the reader is not required to read the original paper to understand the analysis. Next, in subsection IV-B, we analyse the runtime of the algorithm in the worst possible case, then provide examples for how long this would mean in the case of some real-world protocols. Then, in subsection IV-C, we measure the quality of the output of the algorithm when the runtime is bounded, and it is not able to or not allowed to run to completion. Finally, in subsection IV-D, we discuss the effects of the algorithm having to work with incorrect or incomplete message format specifications.

A. Summary of the Algorithm to be Analysed

One of the models that state machine reverse engineering methods can be built on is the teacher-learner model. In this model, there is a teacher that knows the state machine that needs to be inferred, and a learner, whose job is to learn this state machine by communicating with the teacher. The input and output alphabet are known by both parties.

The learner may submit two kinds of queries to the teacher: input queries and equivalence queries. Input queries consist of one or more input characters, to which the teacher responds with the appropriate output sequence. Equivalence queries are sent when the learner believes that it has finished learning the state machine. The teacher then runs several input sequences through both the conjectured and the true state machine. If there is any difference in any of the outputs from the two state machines, the teacher has found a counterexample: a sequence of inputs, for which the true and the inferred state machines behave differently. This counterexample is sent back to the learner, which now needs to refine the conjectured automaton by issuing further input queries. If no difference is found, the algorithm terminates, and the learner is considered successful.

This model works well for learning the state machines of protocols that are used over a computer network. In this case, the input alphabet is made up of the protocol messages that may be sent by clients (i.e. the requests), while the output alphabet comprises the messages that are sent by the server (i.e. the responses). The learner acts as a client, while the teacher acts as a server. However, the teacher typically cannot answer the queries on its own, as it does not know the state machine directly either. The teacher needs to talk to an actual implementation of the protocol (also referred to as a *speaker* of the protocol) and relay requests and replies, rely on previously recorded network traffic to answer the queries, send a cached reply (if caches are used and the same query was seen before), or any combination of these.

The learner may run any state machine inference algorithm that is capable of inferring Mealy machines. A Mealy machine $M(S, S_0, \Sigma, \Lambda, T, G)$ is a state machine where S is a set of states that make up the state machine, $S_0 \in S$ is the initial state, Σ and Λ are the input and output alphabet (respectively), T is the set of transitions that specify the next state if a specified input arrives in the specified state, and finally, G is the output function that defines which character should be output when a specified input arrives in a given state. Less formally, Mealy machines are state machines where the output values are determined not only by the current state, but also by the received input (as opposed to, for example, Moore machines, where the output is only determined by the current state, not the input). For this reason, network protocols can be easily represented as Mealy machines.

In this analysis, we use the algorithm from Székely et al. [6] as this is not only relatively recent, but we also had a working implementation of it. However, our findings in Section IV-D also apply to any algorithm based on a teaching-learning model, and findings in Section IV-C should also apply – with some margin of error – to any variant of Shahbaz and Groz's ${L_M}^+$ algorithm [26] as long as the core logic is left unchanged.

The algorithm starts out knowing nothing about the protocol state machine, but it knows all the possible input and output messages. It maintains an observation table that is filled or appended to based on the requests sent and the replies received.

Based on message type information, data types and field semantics, it generates a smartly chosen subset of every possible protocol message. These are then sent one by one as queries to the teacher. The teacher uses a speaker to answer the queries, resetting the speaker's state machine (e.g. by reconnecting to the server) after each sequence of queries. The learner keeps issuing input queries in rounds, based on the observation table, until the table is considered closed. The table is considered closed when each transition that follows from the observations ends in a state that has already been processed.

When this point is reached, an automaton is constructed based on the information in the observation table, which is then submitted to the teacher in an equivalence query. If the teacher returns a counterexample, the observation table is extended based on the counterexample, more input queries are run until the table is considered closed again, and this process is repeated until no counterexample is found. At that point, the algorithm terminates, the state machine has been successfully learnt. This process is illustrated in Figure 1 for clarity.

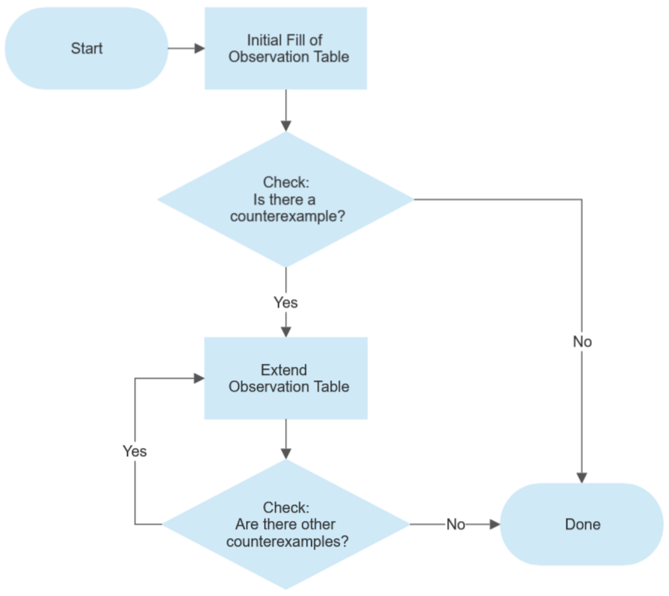
Start
Initial Fill of Observation Table
Check: Is there a counterexample?
Yes
No
Extend Observation Table
Yes
Check: Are there other counterexamples?
No
Done

Fig. 1. The operation of Székely's algorithm in a flowchart.

The messages that are sent by the learner have to be chosen carefully. Sending every possible protocol message is not a viable strategy – considering a very simple protocol where each message contains nothing but a 4-bytes-long integer, there are 2^{32} possibilities. Sending this many messages, possibly multiple times when trying to determine the states of the automaton, would lead to unfeasible running times, large amounts of network traffic, and a heavily redundant state machine. For this reason, the messages are chosen in a smart manner. Since the message types and semantics are assumed to be fully known, this information may be used to generate message instances dynamically on the fly, paying attention to the values in certain fields that might trigger different behaviours. For example, a protocol having a command to read data at a given memory address will return the requested data when a valid memory address is supplied, while it will result in an error message or disconnection if an invalid address is specified. If these cases are not considered and handled properly, the state machine of the implementation (the speaker) could appear to be indeterministic.

B. Runtime, Complexity

As explained in the previous subsection, the algorithm first needs to fill the observation table to the point where it is considered closed. The first row needs $|\Sigma|$ queries to be filled. Then, in each following round, for each element of Σ (there will be $|\Sigma|$ of these), one new request will be issued for each new entry added in the previous round (there will also be $|\Sigma|$ of these), making it a total of $|\Sigma|^2$ requests for a round. However, the length of the request (i.e. how many queries are needed to fulfil it) depends on the length of the requests in the previous round. In the worst case, this length is always the number of the current round, which occurs when the automaton being inferred is maximally deep (see Figure 2 for an example). This follows from the property that there is no shorter path to any of the states in such a state machine than through *all* the previous states. The number of rounds is $|S|+1$ as $|S|$ rounds are needed to find all $|S|$ states, plus 1 to confirm that there are no more.

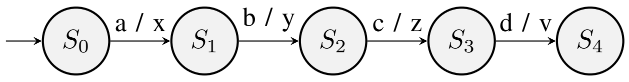
$$\begin{tikzcd} \arrow[r] & S_0 \arrow[r, "a / x"] & S_1 \arrow[r, "b / y"] & S_2 \arrow[r, "c / z"] & S_3 \arrow[r, "d / v"] & S_4 \end{tikzcd}$$

Fig. 2. A maximally deep state machine with 5 states and 4 possible inputs that will result in the worst case number of queries (324 in this case) when filling the observation table.

Based on the above, in the worst case, the number of steps for the initial fill of the observation table is given by Equation 1, which can be rearranged and rewritten through series expansion as Equation 2.

$$q_{l\_ot\_init} = |\Sigma| + \sum_{i=2}^{|S|+1} i * |\Sigma|^2 \tag{1}$$

$$q_{l\_ot\_init} = \frac{1}{2}(|\Sigma|^2 * |S|^2) + \frac{1}{2}(3 * |\Sigma|^2 * |S|) + |\Sigma| \tag{2}$$

Once the observation table is closed, the conjecture is submitted to the teacher. The teacher then needs to run a number of queries against both the conjecture and the speaker. The number of queries is given by Equation 3.

$$q_{t\_conj\_single} = |\Sigma| * sf * n_{tests} \tag{3}$$

In this equation, sf is a stretching factor that is defined as 10 in [6]. The number of tests (n_{tests}) is defined in [23], and is a more complex formula (see Equation 4).

$$n_{tests} = \lceil \frac{1}{\varepsilon} * (\log \frac{1}{\delta} + (\log 2) * (n_{false\_conj} + 1)) \rceil \tag{4}$$

Here, ε and δ are accuracy and confidence parameters, set to 10^{-2} and 10^{-6}, respectively, as per [6]. In addition, $\log$ denotes the natural logarithm function, while n_{false_conj} is the number of conjectures that have been tested previously. This shows that the number of tests also depends on the number of previous (and failed) equivalence queries.

It should be noted that since the test sequences are generated randomly, the number of input queries needed for an equivalence query varies per run of the algorithm, even if the same inputs are used.

The root cause of why the algorithm might make a mistake (and the teacher will find a counterexample) is the automaton having states within reach of each other, with almost identical transitions, as in the case of Figure 3.

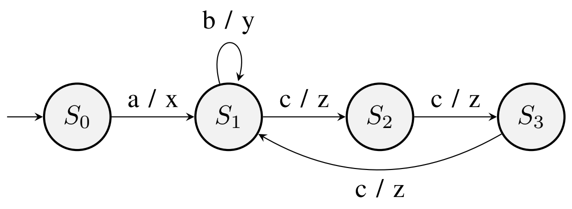
$$\begin{tikzcd} \arrow[r] & S_0 \arrow[r, "a / x"] & S_1 \arrow[loop above, "b / y"] \arrow[r, "c / z"] & S_2 \arrow[r, "c / z"] & S_3 \arrow[ll, bend left, "c / z"] \end{tikzcd}$$

Fig. 3. An example of an automaton where the learner will make a mistake.

Considering only the inputs and outputs of S_2's and S_3's transitions, these states appear to be identical, and the algorithm will make a mistake here. The conjectured (and incorrect) automaton can be seen in Figure 4.

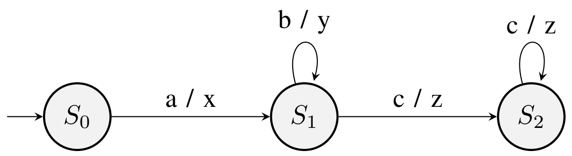
$$\begin{tikzcd} \arrow[r] & S_0 \arrow[r, "a / x"] & S_1 \arrow[loop above, "b / y"] \arrow[r, "c / z"] & S_2 \arrow[loop above, "c / z"] \end{tikzcd}$$

Fig. 4. The incorrect automaton, as conjectured for the first time.

When a counterexample is found, the learner updates its observation table and issues further input queries to close it again. In the worst case, the number of input queries sent is given by Equation 5, which may be rearranged and expanded as Equation 6.

$$q_{l\_ot\_update} = |S|^2 * \sum_{i=|S|}^{|\Sigma|*sf+|S|} i \tag{5}$$

$$q_{l\_ot\_update} = \frac{1}{2} * |S|^2 * (|\Sigma| + sf + 1)(|\Sigma| + 2 * |S| + sf) \tag{6}$$

The maximum number of counterexamples is $|S| - 1$, because the initial state is always found. Thus, the number of steps in Equation 5 will need to be repeated at most $|S| - 1$ times. Furthermore, in the worst case, the total number of counterexample checks is $|S| - 1 + 1$ (that is, $|S|$) because of the final counterexample check that is always performed. From Equations 3 and 4, the total number of input queries required for these can be given as Equation 5.

$$q_{t\_conj\_total} = \sum_{i=0}^{|S|-1} |\Sigma| * 10 * \lceil 10^2 * (\log 10^6 + (\log 2) * (i+1)) \rceil \tag{7}$$

Putting this all together, the number of input queries needed in the absolute worst case is the sum of Equation (1), $(|S| - 1)$ times Equation (4), and Equation (5):

$$q_{total} = q_{l\_ot\_init} + (|S| - 1) * q_{l\_ot\_update} + q_{t\_conj\_total} \tag{8}$$

The complexity of the terms in this equation is $O(|S|^2 * |\Sigma|^2)$, $O(|S|^4 * |\Sigma| + |S|^3 * |\Sigma|^2)$, and $O(|S|^2 * |\Sigma|)$, respectively, with the highest order term being the second one. From this, it follows that the worst-case complexity of this algorithm is $O(|S|^4 * |\Sigma| + |S|^3 * |\Sigma|^2)$.

For the sake of clarity, we have summarized the previous notations and their descriptions in Table I.

TABLE I
NOTATIONS AND THEIR DESCRIPTIONS.

Notation	Meaning
$\|\Sigma\|$	Number of elements in the input alphabet of the state machine of the protocol.
$\|S\|$	Number of states in the state machine of the protocol.
$q_{l\_ot\_init}$	Number of queries the learner needs to close the initial observation table.
$q_{t\_conj\_single}$	Number of queries the teacher needs to test a single conjecture.
$q_{l\_ot\_update}$	Number of queries the learner needs to update its observation table after a counterexample is found by the teacher.
$q_{t\_conj\_total}$	Total number of queries the teacher needs to test every conjecture in the worst possible case.
q_{total}	Total number of queries the learner and the teacher need in the worst possible case.

We were interested to know how many input queries this would mean in the case of real-world protocol automata, and ultimately, how long these would take to be inferred in the absolute worst case. For this reason, we collected information on such protocols. This is shown in Table II.

TABLE II
COMPLEXITY OF SOME REAL-WORLD PROTOCOL AUTOMATA

Protocol	States ($\|S\|$)	Input Msg. Types ($\|\Sigma\|$)	Output Msg. Types ($\|\Lambda\|$)
Modbus/TCP [33]	-	min. 8	min. 8
MQTT [34]	min. 3	7 + 4	7 + 4
BGP [35]	6	4	4
POP3 [36]	4	10	2
IMAP [37]	3	25	15
FTP [38]	min. 8	33	42
MegaD C&C [22]	18	15	13

For Modbus/TCP, there are no protocol states, only TCP states. The typical minimal implementation supports 8 basic commands, but device vendors may add support for additional function codes, including vendor-specific ones. For MQTT, the minimum number of states is 3. This is with a single client, a single topic, and a QoS level of 0. The number of states increases with the number of concurrent clients, topics, and QoS level 2. There are 7 message types that may be sent by either clients or servers, 4 that are only sent by clients, and 4 that are only sent by servers. For POP3 and IMAP, the number of message types may vary per implementation. The table contains the number of message types as per the referenced RFC. For FTP, the number of message types is given based on the referenced RFC. The number of states is based on the work of J. Antunes and N. Neves [39]. Newer versions and custom implementations of FTP support additional message types, and may also have more states. The details of the MegaD botnet's command and control (C&C) protocol were sourced from the paper of Cho et al. [22], who were the first ones to reverse engineer the MegaD botnet's protocols.

Using this information and the above formulas, we calculated the total number of steps required to reverse engineer the state machine of each protocol in the absolute worst case, along with an estimation of how long this would take, assuming that 1000 requests are processed per second[1]. The results are shown in Table III.

TABLE III
NUMBERS OF QUERIES AND THE TIME REQUIRED TO REVERSE ENGINEER REAL-WORLD PROTOCOL STATE MACHINES, ABSOLUTE WORST CASE CALCULATIONS

Protocol	Queries Required (Worst Case)	Time to Recover (Worst Case)
MQTT	508266	8.47 minutes
BGP	425456	7.09 minutes
POP3	637622	10.63 minutes
IMAP	1159434	19.32 minutes
FTP	5101613	85.03 minutes
MegaD C&C	9919884	165.33 minutes

C. Correctness and Completeness for Partial Runs

To gain insight into what level of correctness and completeness is attainable under limitations, we generated 1000 random message format specifications, each having between 5 and 25 input message types, and between 3 and 15 output message types. For each message format specification, we generated 100 state machines having between 3 and 23 states. The number of input and output message types, as well as the number of states were initially chosen based on prior experience with protocols, but were also confirmed using Table II. The message format specifications were generated to resemble those of real network protocols, as described in [18]. We used a variant of the Erdős-Rényi algorithm [40] to generate random graphs with the necessary properties, which were then used as a basis for the protocol state machines. The algorithm had to be modified to ensure that the resulting graph is connected (all protocol states must be reachable), the edges describe the transitions, the transitions occur in a way that each input and output message type is used at least once, and that there is at least one valid transition from each state (which could also be a self-loop). Our implementation is available on Github[2].

Next, we executed the state machine inference algorithm on each of the 100000 scenarios. In all of the cases, the learner, the teacher, and the speaker all had a 100% correct, complete, and concise description of the message format specifications, and only the speaker had initial knowledge of the protocol state machine (this knowledge was 100% correct, complete, and concise). In each scenario, we monitored the execution of the algorithm: we counted the number of queries from the teacher to the speaker, and after each query, we also queried the learner for its current idea of the state machine. This was then compared to the true state machine.

We found that in 83% of the cases, the algorithm finished without finding counterexamples. In these cases, the number of queries required for a 100% complete protocol state machine is, on average, 12% of the total. This also follows from

[1]The exact number of requests that can be sent to and processed by a speaker in a second depends on several factors. It varies by protocol and even by different implementations of the same protocol. The number 1000 was chosen based on prior experience, and is believed to be correct within one order of magnitude, considering non-clustered, networked services.

[2]https://github.com/GergoLadi/StateMachineGenerator

Equations 1 and 3 in Section IV-B. Filling the observation table to the point where it is considered closed requires a number of requests that is generally at least one order of magnitude smaller compared to the number of queries performed while checking for counterexamples. In addition, less complex protocols require a fewer percentage of the total number of queries, while more complex ones require more: a protocol with 10 input message types and 5 states will need 2010 queries to fill the observation table in the worst case scenario, and it will take a further 145100 queries to check for counterexamples. Meanwhile, a protocol with 20 input message types and 10 states will need at most 26020 queries to fill the observation table, and a further 290200 to check for counterexamples. In these examples, the percentage of the queries needed to fill and close the observation table is 1.37% and 8.23%, respectively, in the worst case scenario.

Should these amounts be also considered to be over the limit, it is also possible to ask the algorithm for the inferred state machine between each fill operation, without having to wait for the observation table to be closed. In these cases, the completeness of the inferred state machine is proportional to the number of queries performed during the fill operations, but the relationship is not linear since the inferred state machine is only (and can only be) updated after a fill operation. Queries made after a previous operation do not directly contribute to the completeness score until the end of the current operation. Figure 5 shows an example. The number of fill operations and the number of queries needed for a given operation varies by state machine.

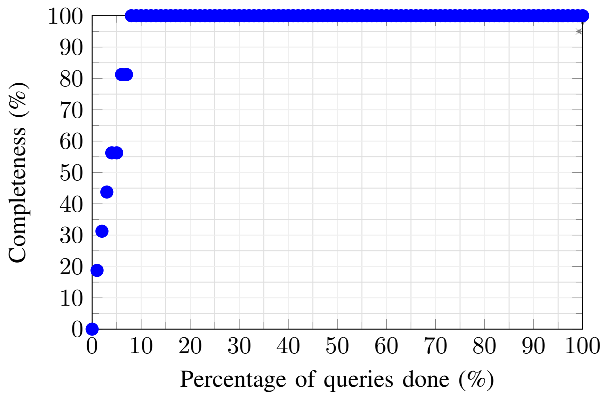

Fig. 5. Completeness in the case of partial runs (example).

In this example, the partial inferred state machine was retrieved and compared to the true state machine between fill operations. Six rounds are required to fill and close the observation table. With each fill operation, more states and transitions are discovered, increasing the completeness score. The first one and the last two rounds improve the score more than the others. The last two rounds require more queries to finish. The whole process makes up less than 10% of the total queries, the rest are issued while looking for a counterexample.

Correctness is always 100% as the observed state machine never contains states or transitions that do not exist in the true state machine.

At least one counterexample was found in the rest of the cases (18%). Since finding a counterexample results in additional queries to fill the observation table again, followed by another round of counterexample checks, we were expecting these cases to require significantly more queries to run to completion. In addition, the queries issued to fill the initial observation table should now make up a smaller percentage of the total.

To confirm our hypothesis, we took the average of all such cases, and looked at the results. We found that the average number of counterexamples was 2.24. The queries needed to fill the initial observation table, on average, make up 1.81% of the total queries. We also found that while the number of queries does increase based on the number of counterexamples found, this increase is generally between 10 and 20 percent per counterexample, far below our expectations. Further analysis showed that the most expensive operation is the final counterexample check that confirms that the inferred state machine is correct and complete, while the other counterexample checks terminate early on, saving a considerable number of queries.

Finding a counterexample also means that there is at least one transition in the inferred state machine that is not correct. Therefore, when analysing these cases, we also considered correctness. The results show that there is a strong correlation between correctness and completeness in these cases, with counterexamples having a lesser degree of effect on correctness than completeness. This is as expected, since the counterexamples stem from single incorrectly assumed transitions, as a result of which, multiple states and their transitions may be missing. Figure 6 illustrates this phenomenon well: considering a concrete example, the first 2 percent of the queries are used to fill the initial observation table. At the end of this phase, over 80% of the state machine is recovered, however, some incorrect transitions were also included, reducing the correctness score. The following 8 percent of the queries are spent looking for counterexamples until one is found. The observation table is extended using this counterexample, eliminating an incorrect transition and discovering real states or transitions, improving both the correctness and the completeness score. The next 12 percent are spent searching for counterexamples again, until one more is found. The observation table is extended again, further improving the scores, then another search for counterexamples begins. Once more, after a further 13 percent of queries, another counterexample is found and is handled accordingly. At this point, all the incorrect transitions have been eliminated, and the entire state machine has been recovered. However, this is not known until the final counterexample check finishes, which takes up the remaining 65 percent of the queries.

In practice, these findings mean that when the number of queries is limited, one may run the algorithm until the observation table is filled and considered closed for the first time, retrieve the current conjecture for the state machine, and consider this the final state machine without running the counterexample checks. In the majority of the cases (82% in our tests), this will be 100% complete and correct, with less than 12% of the queries done. Table IV shows how long some real-world protocol state machines would take to recover using this method compared to the absolute worst case and a more realistic case with a successful first counterexample check. (As

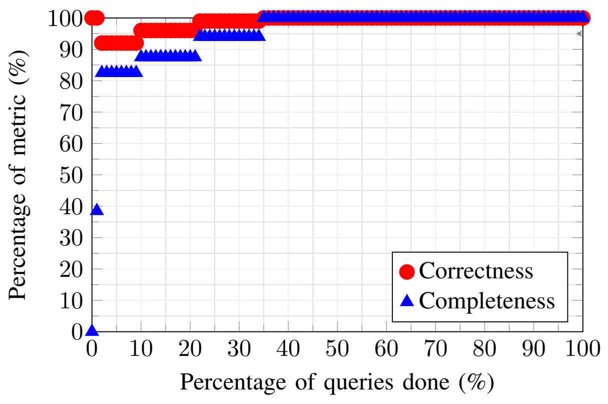

Fig. 6. An example of a run where 3 counterexamples were found.

before, 1000 requests per second are assumed.)

If restrictions allow, counterexample checks should continue running in the background, as even a few checks are better than none. In our experiments, when counterexamples existed, they were generally found within the first 5% of checks, however, this is subject for further research.

D. *Working with Incorrect or Incomplete Information*

As mentioned in subsection IV-A, the FSM recovery algorithm assumes that the message formats it is working with are 100% complete and correct. We have also assumed so in the previous analyses. However, this is usually not the case. For this reason, we conducted an analysis on how the teaching and learning based protocol state machine reverse engineering algorithm handles situations where there is something wrong with the message format specifications. We modified the arrangement from the previous section in order to be able to deterministically inject faults into the message format specifications that are used, repeated the tests, and observed the results. Taking into account the three wider categories of shortcomings regarding the message format specifications as well as the direction of the messages, a total of six cases were examined:

1) Missing client-side message type: For the teaching-learning algorithm, a missing client-side message type is equivalent to missing an element from the input alphabet of the (inferred) FSM. The learner will never send messages of this type to the teacher, who will also never use this message type when verifying the learner's conjectures. The fact that such a message type is missing will not be detected at any point, and the resulting automaton will be missing all the possible transitions for this message type, all the states that are only reachable through one of these missing transitions, and all the transitions from the missing states. Assuming that the true FSM is defined as $M(S, S_0, \Sigma, \Lambda, T, G)$ and the resulting FSM is $M'(S', S_0, \Sigma', \Lambda, T', G)$, where $S' \subseteq S$, $\Sigma' \subseteq \Sigma$, and $T' \subseteq T$, the difference may be formalized as follows:

$T_{miss\_in} = \{t \in T : t(s_i \in S, a \in \Sigma \setminus \Sigma') = s_j \in S\}$

$S_{miss} = \{s \in S : \nexists t_1, t_2, ..., t_n \in T \setminus T_{miss\_in} : t_n(...(t_2(t_1(s_i, a), b), ..., e) = s\} \; (a, b, ..., e \in \Sigma)$

$T_{miss\_out} = \{t \in T \setminus T_{miss\_in} : t(s_i \in S_{miss}, a \in \Sigma) = s_j \in S\}$

Here, T_{miss_in} is the set of transitions that would be triggered by a missing element of the input alphabet; S_{miss} is the set of states that are not reachable as no path exists to them without the transitions in T_{miss_in}; and T_{miss_out} is the set of transitions that would be triggered by a known (not missing) element of the input alphabet, but originate in unreachable states. Knowing these sets, the graph edit distance between M and M' may be given using the formula:

$$GED(M, M') = |T_{miss\_in}| + |S_{miss}| + |T_{miss\_out}| \qquad (9)$$

Graph Edit Distance (GED) [41] is a measure of similarity between two graphs. Since state machines can be represented as graphs, it is possible to compare a true and an inferred state machine using GED. A distance of 0 means that the two graphs are isomorphic, thus the inferred state machine behaves the same way as the true one.

While the formula assumes that the true FSM (M) is known, this is rarely the case. However, it may still be used to estimate the difference between M and M' if some information is known about the quality of the reverse engineered message format specifications (e.g. the expected number of missing message types) and the properties of the protocol state machine (e.g. the density of the state machine's graph). In addition, when the true FSM is known, the formula proves to be useful for the analysis of the effects of (injected) faults in the message format specifications.

A missing client-side message type does not have an effect on correctness or conciseness, but completeness is negatively affected.

2) Missing server-side message type: This is similar to the previous issue in that entire message types may be missing. In this case, a server-side message type is missing, such as an indication of an error for a failed authentication attempt. (This could be the result of no network traces containing sign-in attempts with an incorrect password.)

For the algorithm, this case is equivalent to missing an element from the output alphabet of the FSM. The teacher will detect this the first time an unknown message type is received as a response from the actual server implementation.

Since having message format specifications that are as correct and complete as possible is paramount, the state machine reverse engineering process should be stopped, and the newly discovered message type should be analysed, understood, and added to the message format specification. Only then should the state machine reverse engineering process be restarted.

3) Duplicate client-side message type: In this case, there is at least one true message type that was identified as two (or more) different message types in the message format specifications. For example, this may happen if a flag field precedes the operation code field in the message header, and the flag field is incorrectly assumed to be a part of the operation code field. Under these circumstances, inference algorithms might identify each true message type as two different message types if the messages sometimes appear with the flag set and sometimes unset.

For the teaching-learning algorithm, this is equivalent to having two (or more) almost identical copies of the same element in the input alphabet of the FSM. The learner and the

TABLE IV
NUMBER OF QUERIES REQUIRED AND TIME TO RECOVER FOR REAL-WORLD PROTOCOL STATE MACHINES, ASSUMING NO COUNTEREXAMPLES AND SKIPPING COUNTEREXAMPLE CHECKS.

Protocol	**Queries Required**	**Time to Recover**	**Percent of Worst Case**	**Percent of a More Realistic Case**
MQTT	1100	1.10 seconds	0.22 %	0.68 %
BGP	436	0.43 seconds	0.10 %	0.75 %
POP3	1410	1.41 seconds	0.22 %	0.96 %
IMAP	5650	5.65 seconds	0.49 %	1.53 %
FTP	47949	47.95 seconds	0.94 %	9.10 %
MegaD C&C	42540	42.54 seconds	0.43 %	16.35 %

teacher will both use the duplicate message types, and for each transition triggered by these elements, the resulting automaton will have two (or more) practically identical transitions: one for each copy of the element.

While this does not adversely affect the correctness and coverage of the result, it does impact conciseness. Assuming that the algorithm runs to completion, and that there were no missing or incorrect client-side message types, this impact may be calculated as follows:

$$\Delta_{conciseness} = 1 - \frac{|T'|}{|T|} \tag{10}$$

where T is the set of transitions in the true protocol state machine, and T' is the set of transitions in the inferred protocol state machine. An impact of 0 means that there were no duplicate client-side messages, while an impact of 1 means that there are twice as many transitions in the inferred protocol state machine due to message type duplication. For example, in a state machine with 5 client-side message types with 2 transitions each, having 2 duplicate message types results in 4 extra transitions, resulting in an impact of 0.4.

Duplicate client-side message types may be remedied by employing better message format inference algorithms or having higher quality input (e.g. better network traces). Alternatively, duplicate message types may also be detected by inspecting the resulting protocol state machine: for any pair of inputs, if all the transitions (and all the outputs for the transitions) are the exact same, the pair of message types should be marked for further inspection.

4) Duplicate server-side message type: Just as a single client-side message type might end up getting identified as two (or more) different message types in the specifications, the same may happen to server-side message types, and for the same reason.

For the state machine inference algorithm, this is equivalent to having two (or more) slightly different copies of the same element in the output alphabet of the FSM. Suppose that in the true specifications, there exists a server-side message type x that is sent in response to a client-side message type a. In the inferred specifications, x appears as y and z; for example, because a flag field was considered to be part of the operation code field. The teacher and the learner will classify responses as y or z, while the software that is running the implementation will operate according to the true specification.

Due to this discrepancy, it will eventually happen that the teacher sends query a in the same state twice, and once it receives y, once z as a response. This can and will be detected as indeterministic behaviour. If this happens, the state machine reverse engineering process cannot continue; the message format specifications need to be reconciled first.

5) Incorrect client-side message type: Apart from being missing and duplicate, message types may also be incorrect, i.e. not matching the true specifications. Depending on what exactly is incorrect, the effects range from nothing to a heavily incomplete protocol state machine.

Consider a field in one of the requests that is supposed to be a transaction identifier and is expected to be a random value in each request. If this field appears incorrectly as a constant of some value in the reverse engineered specifications, the same value will be used in each request, but requests will complete successfully, and this will have no effect on the characteristics of the resulting protocol state machine.

On the other hand, if, for example, a field that is supposed to be a sequence number is identified as a constant (or random) value in the reverse engineered specifications, the server will consider messages from the teacher-learner algorithm to be out of sequence, and either respond with an error message to every request or terminate the connection. This situation can be detected, and the message format specifications can be fixed, but in more subtle cases, only a subset of the message types will be affected, and this may remain unnoticed, resulting in an incomplete protocol state machine. In these cases, the effect of incorrect client-side message types is similar to that of missing client-side message types: transitions triggered by these message types will not be present in the protocol state machine, and neither will be states that are only reachable through these, nor transitions that originate in these missing states.

6) Incorrect server-side message type: Not only client-side message types may be incorrect, so may be server-side message types. As long as the teacher can identify which message type a given received message corresponds to (failing that, this is no longer a case of just an incorrect message type), inconsistencies found between the true and the inferred message format specifications may be ignored. However, it would be preferable for the algorithm to make notes of these, as this information may be used to improve the message format specifications.

For example, in the inferred message format specifications, there could be a field that was identified to be a random value in each message, but by looking at the responses of the server, it might become apparent that this is a counter that is always increased by one. The message format specifications can then be updated accordingly.

Based on the above, it can be stated that the different problems related to the different message types have varying effects on the resulting protocol state machine. As summarized in Table V, the most problematic issue is the missing client-side message type, as there is no way to detect that this is the case, and this prevents the algorithm from recovering the entire true protocol state machine. The second worst issue is the presence of incorrect client-side message types; while these can be detected and fixed in some cases, in the others, these will also prevent the algorithm from recovering the entire true protocol state machine. In all the other cases, the presence of the issue will always be detected, and steps can be taken to remedy the situation.

It is also worth noting that fixing these issues also improves the quality of the inferred message format specifications. As a consequence, if a live speaker of the protocol is available, it may be worth running the protocol state machine reverse engineering algorithm even if we are only interested in the message format specifications since the algorithm will be able to detect and pinpoint some mistakes and inconsistencies in the message formats.

V. Conclusion

In this paper, we have reviewed the means and methods of the evaluation of the performance of protocol state machine reverse engineering methods, including commonly used metrics of quality, and runtime and complexity. We have proposed new performance metrics related to bounded runtime, and shown the importance of analysing the effects of incomplete or incorrect input.

Applying these existing and new methods, we have analysed the teaching and learning based protocol state machine reverse engineering method of Székely et al. We have given a formula for the worst case complexity of the algorithm and calculated how long real-world protocols would take to have their state machines reverse engineered. We have shown that when runtime is limited, as few as 12% of the total queries may be enough to recover a 100% correct and complete protocol state machine for a realistic protocol (however, one can never be sure that this is the case until all of the queries have been performed). We have also analysed how the algorithm handles cases where the message type specifications are not 100% correct, complete, and concise, as the algorithm originally requires. The results indicate that the different classes of problems affect the reverse engineered protocol state machine in different ways, and that the direction of the problematic message type also matters. For example, missing client-side messages type cannot be detected, and will result in missing states and transitions in the state machine, while missing or duplicate server-side message types will be detected, and will cause the algorithm to halt without producing a state machine.

In the future, we plan to analyse other protocol state machine reverse engineering methods following the same process and compare the results. Unfortunately, concrete implementations are still rarely published, making this difficult to achieve.

Acknowledgment

The authors would like to thank Dr. Levente Buttyán for his advice and comments that helped improve this paper.

This research was supported by the Hungarian National Research, Development and Innovation Fund (NKFIH, project no. 2017-1.3.1-VKE-2017-00029), and by the Ministry of Innovation and Technology NRDI Office within the framework of the Artificial Intelligence National Laboratory Program.

References

[1] J. Antunes, N. Neves, M. Correia, P. Verissimo, and R. Neves, "Vulnerability discovery with attack injection," *IEEE Transactions on Software Engineering*, vol. 36, no. 3, pp. 357–370, May 2010. DOI: 10.1109/TSE.2009.91 Generated from Scopus record by KAUST IRTS on 2021-03-16.

[2] H. J. Wang, C. Guo, D. R. Simon, and A. Zugenmaier, "Shield: Vulnerability-driven network filters for preventing known vulnerability exploits," *SIGCOMM Comput. Commun. Rev.*, vol. 34, no. 4, p. 193–204, aug 2004. DOI: 10.1145/1030194.1015489

[3] T. Krueger, H. Gascon, N. Krämer, and K. Rieck, "Learning stateful models for network honeypots," in *Proceedings of the 5th ACM Workshop on Security and Artificial Intelligence*, ser. AISec '12. New York, NY, USA: Association for Computing Machinery, 2012. DOI: 10.1145/2381896.2381904. ISBN 9781450316644 p. 37–48.

[4] J. Narayan, S. K. Shukla, and T. C. Clancy, "A survey of automatic protocol reverse engineering tools," *ACM Computing Surveys*, vol. 48, no. 3, pp. 1–26, dec 2015. DOI: 10.1145/2840724

[5] J. Duchêne, C. Le Guernic, E. Alata, V. Nicomette, and M. Kaâniche, "State of the art of network protocol reverse engineering tools," *Journal of Computer Virology and Hacking Techniques*, vol. 14, no. 1, pp. 53–68, Feb 2018. DOI: 10.1007/s11416-016-0289-8

[6] G. Székely, G. Ládi, T. Holczer, and L. Buttyán, "Protocol state machine reverse engineering with a teaching-learning approach," *Acta Cybernetica*, vol. 25, no. 2, pp. 517–535, August 2021. DOI: 10.14232/acta-cyb.288213

[7] D. Lee and M. Yannakakis, "Principles and methods of testing finite state machines – A survey," *Proceedings of the IEEE*, vol. 84, no. 8, pp. 1090–1123, 1996. DOI: 10.1109/5.533956

[8] T. Andrew, "How samba was written," https://download.samba.org/pub/tridge/misc/french_cafe.txt, pp. 1–2, 2003.

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

[10] W. Cui, V. Paxson, N. C. Weaver, and Y. H. Katz, "Protocol-independent adaptive replay of application dialog," in *Network and Distributed System Security Symposium 2006*, 2006, pp. 1–15.

[11] W. Cui, J. Kannan, and H. J. Wang, "Discoverer: Automatic protocol reverse engineering from network traces," in *16th USENIX Security Symposium (USENIX Security 07)*. Boston, MA: USENIX Association, Aug. 2007, pp. 1–14. [Online]. Available: https://www.usenix.org/conference/16th-usenix-security-symposium/discoverer-automatic-protocol-reverse-engineering-network

[12] Y. Wang, X. Li, J. Meng, Y. Zhao, Z. Zhang, and L. Guo, "Biprominer: Automatic mining of binary protocol features," in *2011 12th International Conference on Parallel and Distributed Computing, Applications and Technologies*, 2011. DOI: 10.1109/PDCAT.2011.25 pp. 179–184.

[13] J. Antunes, N. F. Neves, and P. Verissimo, "ReverX: Reverse engineering of protocols," *12th International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT)*, pp. 1–25, 2011.

[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)*, pp. 1–10, 2012. DOI: 10.1109/ICNP.2012.6459963

[15] J.-Z. Luo and S.-Z. Yu, "Position-based automatic reverse engineering of network protocols," *Journal of Network and Computer Applications*, vol. 36, no. 3, pp. 1070–1077, 05 2013. DOI: 10.1016/j.jnca.2013.01.013

[16] G. Bossert, F. Guihéry, and G. Hiet, "Towards Automated Protocol Reverse Engineering Using Semantic Information," in *ASIA CCS '14*, Kyoto, Japan, Jun. 2014. DOI: 10.1145/2590296.2590346 pp. 51–62, 12 pages.

TABLE V
A SUMMARY OF THE POSSIBLE PROBLEMS WITH MESSAGE TYPES, THEIR DETECTABILITY, AND THEIR EFFECTS.

Problem	**Message type**	**Can be detected?**	**Effect on the protocol state machine**
Missing	client-side	No	Missing states and transitions (worse completeness score)
Missing	server-side	Yes	No state machine is produced
Duplicate	client-side	Yes	Duplicate transitions (worse conciseness score)
Duplicate	server-side	Yes	No state machine is produced
Incorrect	client-side	In some cases	None to missing states and transitions
Incorrect	server-side	Yes	None

[17] I. Bermudez, A. Tongaonkar, M. Iliofotou, M. Mellia, and M. M. Munafò, "Towards automatic protocol field inference," *Computer Communications*, vol. 84, pp. 40–51, 2016. DOI: https://doi.org/10.1016/j.comcom.2016.02.015

[18] G. Ládi, L. Buttyán, and T. Holczer, "GrAMeFFSI: Graph analysis based message format and field semantics inference for binary protocols using recorded network traffic," *Infocommunications Journal*, vol. 12, no. 2, pp. 25–33, August 2020. DOI: 10.36244/ICJ.2020.2.4

[19] X. Wang, K. Lv, and B. Li, "Ipart: an automatic protocol reverse engineering tool based on global voting expert for industrial protocols," *International Journal of Parallel, Emergent and Distributed Systems*, vol. 35, no. 3, pp. 376–395, 2020. DOI: 10.1080/17445760.2019.1655740

[20] Y. Huang, H. Shu, F. Kang, and Y. Guang, "Protocol reverse-engineering methods and tools: A survey," *Computer Communications*, vol. 182, pp. 238–254, 2022. DOI: https://doi.org/10.1016/j.comcom.2021.11.009

[21] C. Leita, K. Mermoud, and M. Dacier, "Scriptgen: an automated script generation tool for honeyd," in *21st Annual Computer Security Applications Conference (ACSAC'05)*, 2005. DOI: 10.1109/CSAC.2005.49 pp. 203–214.

[22] C. Y. Cho, D. Babić, E. C. R. Shin, and D. 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*, ser. CCS '10. New York, NY, USA: Association for Computing Machinery, 2010. DOI: 10.1145/1866307.1866355. ISBN 9781450302456 p. 426–439.

[23] D. Angluin, "Learning regular sets from queries and counterexamples," *Information and Computation*, vol. 75, no. 2, pp. 87–106, 1987. DOI: https://doi.org/10.1016/0890-5401(87)90052-6

[24] Y. Wang, Z. Zhang, D. D. Yao, B. Qu, and L. Guo, "Inferring protocol state machine from network traces: A probabilistic approach," in *Applied Cryptography and Network Security*, J. Lopez and G. Tsudik, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011. DOI: 10.1007/978-3-642-21554-4_1. ISBN 978-3-642-21554-4 pp. 1–18.

[25] M.-M. Xiao and Y.-P. Luo, "Automatic protocol reverse engineering using grammatical inference," *Journal of Intelligent & Fuzzy Systems*, vol. 32, pp. 3585–3594, 2017. DOI: 10.3233/JIFS-169294 5.

[26] M. Shahbaz and R. Groz, "Inferring mealy machines," in *FM 2009: Formal Methods*, A. Cavalcanti and D. R. Dams, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009. ISBN 978-3-642-05089-3 pp. 207–222.

[27] P. M. Comparetti, G. Wondracek, C. Kruegel, and E. Kirda, "Prospex: Protocol specification extraction," in *2009 30th IEEE Symposium on Security and Privacy*, 2009. DOI: 10.1109/SP.2009.14 pp. 110–125.

[28] C. Y. Cho, D. Babić, P. Poosankam, K. Z. Chen, E. X. Wu, and D. Song, "MACE: Model-inference-Assisted concolic exploration for protocol and vulnerability discovery," in *20th USENIX Security Symposium (USENIX Security 11)*. San Francisco, CA: USENIX Association, Aug. 2011, pp. 1–16. [Online]. Available: https://www.usenix.org/conference/usenix-security-11/mace-model-inference-assisted-concolic-exploration-protocol-and

[29] C. Yang, C. Fu, Y. Qian, Y. Hong, G. Feng, and L. Han, "Deep learning-based reverse method of binary protocol," in *Security and Privacy in Digital Economy*, S. Yu, P. Mueller, and J. Qian, Eds. Singapore: Springer Singapore, 2020. DOI: 10.1007/978-981-15-9129-7_42. ISBN 978-981-15-9129-7 pp. 606–624.

[30] Y. Wang, B. Bai, X. Hei, L. Zhu, and W. Ji, "An unknown protocol syntax analysis method based on convolutional neural network," *Transactions on Emerging Telecommunications Technologies*, vol. 32, no. 5, p. e3922, 2021. DOI: https://doi.org/10.1002/ett.3922

[31] Y. Ye, Z. Zhang, F. Wang, X. Zhang, and D. Xu, "Netplier: Probabilistic network protocol reverse engineering from message traces," 2021.

[32] B. Ning, X. Zong, K. He, and L. Lian, "Preiud: An industrial control protocols reverse engineering tool based on unsupervised learning and deep neural network methods," *Symmetry*, vol. 15, no. 3, pp. 1–22, 2023. DOI: 10.3390/sym15030706

[33] Modbus Organization, Inc., "Modbus application protocol specification v1.1b3," pp. 1–50, 2012. [Online]. Available: http://www.modbus.org/docs/Modbus_Application_Protocol_V1_1b3.pdf

[34] A. Banks, E. Briggs, K. Borgendale, and R. Gupta, "MQTT Version 5.0 (OASIS Standard)," pp. 1–120, 2019. [Online]. Available: https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.html

[35] Y. Rekhter, S. Hares, and T. Li, "A Border Gateway Protocol 4 (BGP-4)," RFC 4271, pp. 1–104, Jan. 2006.

[36] M. Rose, "Post Office Protocol - Version 3," RFC 1081, pp. 1–16, Nov. 1988.

[37] M. Crispin, "Internet Message Access Protocol - Version 4rev1," RFC 3501, pp. 1–108, Mar. 2003.

[38] J. Postel, "File Transfer Protocol," RFC 765, pp. 1–70, Jun. 1980.

[39] J. Antunes and N. Neves, "Automatically complementing protocol specifications from network traces," in *Proceedings of the 13th European Workshop on Dependable Computing*, ser. EWDC '11. New York, NY, USA: Association for Computing Machinery, 2011. DOI: 10.1145/1978582.1978601. ISBN 9781450302845 p. 87–92.

[40] P. Erdős and A. Rényi, "On random graphs I." *Publicationes Mathematicae Debrecen*, vol. 6, pp. 290–297, 1959.

[41] A. Sanfeliu and K.-S. Fu, "A distance measure between attributed relational graphs for pattern recognition," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. SMC-13, no. 3, pp. 353–362, 1983. DOI: 10.1109/TSMC.1983.6313167

Gergő Ládi was born in Hungary in 1990. He received his Master's degree in Computer Science Engineering in 2018 from Budapest University of Technology and Economics, Hungary, where he is currently pursuing his Ph.D. degree with the Laboratory of Cryptography and System Security. His main areas of research are protocol reverse engineering automation, cloud security, and the security of operating systems.

Tamás Holczer was born in 1981 in Budapest. He received the Ph.D. degree in Computer Science from the Budapest University of Technology and Economics (BME) in 2013. Since 2013 he has been working as an assistant professor in the Laboratory of Cryptography and System Security (CrySyS), Department of Telecommunications, Budapest University of Technology and Economics. Fields of interest: In the past his research interests and his Ph.D. dissertation were focused on the privacy problems of wireless sensor networks and ad hoc networks. Lately he is working on the security aspects of cyber physical systems. The research topics include: security of industrial control networks, honeypot technologies in embedded systems, network monitoring and intrusion detection in industrial networks, and security aspects of intra-vehicular networks.