Using Network Traces to Generate Models for Automatic Network Application Protocols Diagnostics

Martin Holkovič[1], Ondřej Ryšavý[2] and Libor Polčák[2]
[1]CESNET a.l.e., Zikova 1903/4, 160 00 Prague, Czech Republic
[2]Faculty of Information Technology, Brno University of Technology, Bozetechova 1/2, 612 66 Brno, Czech Republic
holkovic@cesnet.cz, rysavy@fit.vutbr.cz and polcak@fit.vutbr.cz

Keywords: Network diagnostics, automatic diagnostics, protocol model from traces.

Abstract: Network diagnostics is a time-consuming activity that requires an administrator with good knowledge of network principles and technologies. Even if some network errors have been resolved in the past, the administrator must spend considerable time removing these errors when they reoccur. This article presents an automated tool to learn the expected behavior of network protocols and possible variations. The created model can be used to automate the diagnostic process. The model presents a finite automaton containing protocol behavior for different situations. Diagnostics of unknown communication is performed by checking the created model and searching for error states and their descriptions. We have also created a proof-of-concept tool that demonstrates the practical potential of this approach.

1 INTRODUCTION

Computer networks consist of a large number of devices and applications that communicate with each other. Due to the complexity of the network, errors occurred on a single device can negatively affect the network services and thus the user experience. There are various sources of error, such as misconfiguration, poor connectivity, hardware error, or even user misbehavior. End users are often unable to solve these problems and seek help from a network administrator. The administrator must diagnose the current situation, find the cause of the problem, and correct it to provide the service again.

The administrator diagnoses problems by checking communication and finding possible causes for these errors. Troubleshooting can be a rather complex activity requiring good technical knowledge of each network entity. Another complication is that the administrator often has to check the number of possible causes to find the true source of the problem, which requires some time. Network problems reappear even after the administrator detects and resolves these issues, such as repeating the same user error or when the application update on the server changes the expected behavior. All these problems make the diagnostic process a time-consuming and challenging activity that requires much administrator attention.

(Zeng et al., 2012) provides a short survey that shows that network diagnostics is time-consuming, and administrators wish to have a more sophisticated diagnostic tool available. Since each environment is different, the use of universal tools is difficult. It would be useful to have a tool that adapts to behavior on a particular network. The tool should learn the behavior of the network itself without the need to program or specify rules on the behavior of individual communicating applications and services.

Our goal is to develop a tool that automatically creates a protocol behavior model. We are not aiming at creating a general model for use in all networks, but the model should describe diagnosed network only. Instead of writing the model manually, the administrator provides examples (traces) of the protocol conversations in the form of PCAP files. The administrator provides two groups of files. The first group contains traces of normal behavior, while the second group consists of known, previously identified error traces. Based on these groups, the tool creates a protocol model. When the model is created, it can be used for detection and diagnosis of issues in the observed network communication. Once the model is created, additional traces may be used to improve the model gradually.

Our focus is on detecting application layer errors in enterprise networks. Thus, in the presented work, we do not consider errors occurred on other layers, e.g., wireless communication (Samhat et al., 2007), routing errors (Dhamdhere et al., 2007), or performance issue on the network layer (Ming Luo, 2011). Because we are focusing on enterprise networks, we make some assumption on the availability of required source data. We expect that administrators using this approach have access to network traffic as the same administrators operate the network infrastructure, and it is possible to provide enough visibility to data communication. Even the communication outside the company's network is encrypted, the traffic between the company's servers and inside the network is many times unencrypted, or the data can be decrypted by providing server's private key or logging the symmetric session key [1]. Also, the source capture files have no or minimal packet loss. An administrator can recapture the traffic if necessary.

When designing the system, we assumed some practical considerations:

- no need to implement custom application protocol dissectors;
- application error diagnostics cannot be affected by lower protocols (e.g., version of IP protocol, data tunneling protocol);
- easily readable protocol model - the created model can be used for other activities too (e.g., security analysis).

To demonstrate the potential of our approach, we have created and evaluated a proof-of-concept implementation available as a command line tool.

The main contribution of this paper is a new automatic diagnostic method for error detection in network communication of commonly used application protocols. The method creates a protocol behavior model from packets traces that contain both correct and error communication patterns. The administrator can also use the created model for documentation purposes and as part of a more detailed analysis,e.g., performance or security analysis.

This paper is organized as follows: Section 2 describes existing work comparable to the presented approach. Section 3 overviews the system architecture. Section 4 provides details on the method, including algorithms used to create and use a protocol model. Section 5 presents the evaluation of the tool implementing the proposed system. Finally, Section 6 summarizes the paper and identifies possible future work.

[1] http://www.root9.net/2012/11/ssl-decryption-with-wireshark-private.html

2 RELATED WORK

Recently published survey paper (Tong et al., 2018) divides issues related to network systems as either *application-related* or *network-related* problems. Notable attention in troubleshooting of network applications was concentrated on networked multimedia systems, e.g., (Leaden, 2007), (Shiva Shankar and Malathi Latha, 2007), (Luo et al., 2007). Multimedia systems require that certain quality of service (QoS) is provided by the networking environment otherwise various types of issues can occur. Network issues comprise network reachability problems, congestion, excessive packet loss, link failures, security policy violation, and router misconfiguration.

Traditionally, network troubleshooting is a mostly manual process that uses several tools to gather and analyze relevant information. The ultimate tool for manual network traffic analysis and troubleshooting is Wireshark (Orzach, 2013). It is equipped with a rich set of protocol dissectors that enables to view details on the communication at different network layers. An administrator has to manually analyze the traffic and decide which communication is abnormal, possibly contributing to the observed problem. Though Wireshark offers advanced filtering mechanism, it lacks any automation (El Sheikh, 2018).

Network troubleshooting employs active, passive, or hybrid methods (Traverso et al., 2014). Active methods rely on the tools that generate probing packets to locate network issues (Anand and Akella, 2010). Specialized tools using generated diagnostic communication were also developed for testing network devices (Procházka et al., 2017). Contrary to active methods, the passive approach relies only on the observed information. Various data sources can be mined to obtain enough evidence to identify the problem. The collected information is then evaluated by the troubleshooting engine. The engine can use different techniques of fault localization.

Rule-based systems describe normal and abnormal states of the system. The set of rules is typically created by an expert and represents the domain knowledge for the target environment. Rule-based systems often do not directly learn from experience. They are also unable to deal with new previously unseen situations, and it is hard to maintain the represented knowledge consistently (łgorzata Steinder and Sethi, 2004).

Statistical and machine learning methods were considered for troubleshooting misconfigurations in home networks (Aggarwal et al., 2009) and diagnosis of failures in large Internet sites (Chen et al., 2004). *Tranalyzer* (Burschka and Dupasquier, 2017)

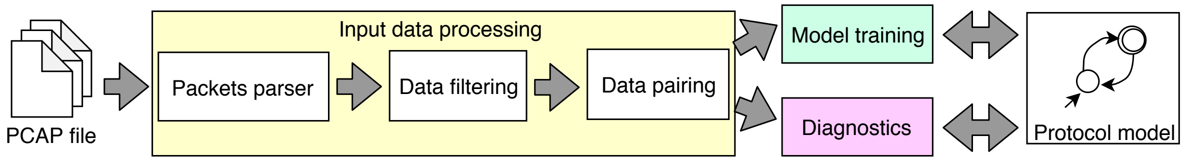

Figure 1: After the system processes the input PCAP files (the first yellow stage), it uses the data to create the protocol behavior model (the second green stage) or to diagnose an unknown protocol communication using the created protocol model (the-third purple stage).

is a flow-based traffic analyzer that performs traffic mining and statistical analysis enabling troubleshooting and anomaly detection for large-scale networks. Big-DAMA (Casas et al., 2016) is another framework for scalable online and offline data mining and machine learning supposed to monitor and characterize extremely large network traffic datasets.

Protocol analysis approach attempts to infer a model of normal communication from data samples. Often, the model has the form of a finite automaton representing the valid protocol communication. An automatic protocol reverse engineering that stores the communication patterns into regular expressions was suggested in (Xiao et al., 2009). Tool *ReverX* (Antunes et al., 2011) automatically infers a specification of a protocol from network traces and generates corresponding automaton. Recently, reverse engineering of protocol specification only from recorded network traffic was proposed to infer protocol message formats as well as certain field semantics for binary protocols (Lodi et al., 2018).

3 SYSTEM ARCHITECTURE

This section describes the architecture of the proposed system which learns from communication examples and diagnoses unknown communications. The system takes PCAP files as input data, where one PCAP file contains only one complete protocol communication. An administrator marks PCAP files as correct or faulty communication examples before model training. The administrator marks faulty PCAP files with error description and a hint on how to fix the problem. The system output is a model describing the protocol behavior and providing an interface for using this model for the diagnostic process. The diagnostic process takes a PCAP file with unknown communication and checks whether this communication contains an error and if yes, returns a list of possible errors and fixes.

The architecture, shown in Figure 1, consists of multiple components, each implementing a stage in the processing pipeline. The processing is staged as follows:

- **Input data processing** - Preprocessing is responsible for converting PCAP files into a format suitable for the next stages. Within this stage, the input packets are decoded using protocol parser. Next, the filter is applied to select only relevant packets. Finally, the packets are grouped to pair request to their corresponding responses.
- **Model training** - The training processes several PCAP files and creates a model characterizing the behavior of the analyzed protocol. The output of this phase is a protocol model.
- **Diagnostics** - In the diagnostic component, an unknown communication is analyzed and compared to available protocol models. The result is a report listing detected errors and possible hints on how to correct them.

In the rest of the section, the individual components are described in detail. Illustrative examples are provided for the sake of better understanding.

3.1 Input Data Processing

This stage works directly with PCAP files provided by the administrator. Each file is parsed by *TShark* [2] which exports decoded packets to JSON format. The system further processes the JSON data by filtering irrelevant records and pairs request packets with their replies. The output of this stage is a list of tuples representing atomic transactions.

3.1.1 Packets parser

Instead of writing our packet decoders, we use the existing implementation provided by *TShark*. TShark is the console version of the well-known Wireshark protocol analyzer which supports many network protocols and can also analyze tunneled and fragmented packets. In the case the Wireshark does not support some protocol, e.g., proprietary, it is possible to use a tool which generates dissectors from XML files (Golden and Coffey, 2015). The system converts each input PCAP file into the JSON format (TShark supports multiple formats). The JSON

[2] https://www.wireshark.org/docs/man-pages/tshark.html

```
...
"eth": {
  "eth.dst": "f0:79:59:72:7c:30",
  "eth.type": "0x00000800",
  ...
},
...
"dns": {
  "dns.id": "0x00007956",
  "dns.flags.response": "0",
  "dns.flags.opcode": "0",
  "dns.qry.name": "mail.patriots.in",
  ...
},
...
```

Figure 2: Excerpt from the TShark output into JSON format. The JSON represents POP3 packet values from all network protocols in a key-value data format.

format represents data as a key-value structure (see Figure 2), where the key is the name of the protocol field according to Wireshark definition [3], e.g., *pop.request.command*.

3.1.2 Data filtering

The JSON format from the TShark output is still protocol dependent because the field names are protocol-dependent, and we have to know which key names each protocol uses. The system converts the data into a more generic format to make the next processing protocol independent. We have found out, that most of the application protocols use a request-reply communication pattern. The system filters requests and replies from each protocol and removes the rest of the data. Even though the protocols use the same communication pattern, they use a different naming convention to mark the reply and response values (see Table 1).

The problem is how to find the requests and replies in the JSON data. In our solution, we have created a database of protocols and their processed field names. In the case the protocol is not yet in the database, we require the administrator to add these two field names to the database. The administrator can get the field names from the Wireshark tool easily by clicking on the appropriate protocol field.

Messages can also contain additional information and parameters, e.g., server welcome message. The system also removes this additional information to allow generalization of otherwise different messages during the protocol model creation. For example, the welcome server message often contains the current date and time, which is always different, and these different messages would create a lot of unrepeatable protocol states. The Figure 3 shows the resulting format from the *Data filtering* step.

[3] https://www.wireshark.org/docs/dfref/

Table 1: Several application protocols with their request and reply field names with example values. The system takes only data from these fields and drops the rest.

Name	Type	Field name	E.g.
SMTP	Request	smtp.req.command	MAIL
	Reply	smtp.response.code	354
FTP	Request	ftp.request.command	RETR
	Reply	ftp.response.code	150
POP	Request	pop.request.command	STAT
	Reply	pop.response.indicator	+OK
DNS	Request	dns.flags.opcode	0
	Reply	dns.flags.rcode	0

Unfortunately, TShark marks some unpredictable data (e.g., authentication data) in some protocols as regular requests and does not clearly distinguish it. These values are a problem in later processing because these unpredictable values create ungeneralizable states during the protocol model learning phase. In our tests, we have observed that regular requests have a maximal length of 6 characters, and unpredictable requests have a much longer length. Based on our finding, the system drops all requests that are longer than six characters, and if necessary, the network administrator can change this value. Distinguishing of these unpredictable requests also for other protocols should be more focused in future research.

3.1.3 Data pairing

To avoid pairing requests and replies during the protocol model learning process, the system pairs each reply to its request in the data processing stage. After this pairing process, the system represents each protocol with a list of pairs, each pair containing one request and one reply. This pairing also simplifies the model learning process because some protocols use the same reply value for multiple commands (e.g., POP3 reply *"+OK"* for all correct replies) and the model could improperly merge several independent reply values into one state (e.g., all POP3 success replies jumps into one state).

The pairing algorithm iteratively takes requests one by one and chooses the first reply it finds. If no reply follows, the system pairs the request with a *"NONE"* reply. In some protocols, the server sends a reply to the client immediately after they connect to the server. E.g., the POP server informs the clients if it can process their requests or not. We have solved this problem by pairing these replies with the empty request *"NONE"*. The result of the pairing process is a sequence of pairs, where each pair consists of one request and one reply. The Figure 3 shows an example of this pairing process.

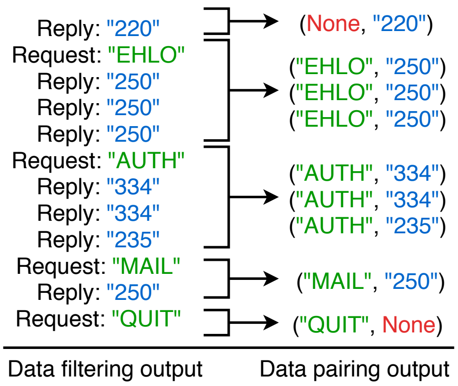

Figure 3: Example of an SMTP communication in which the client authenticates, sends an email and quits the communication. The left part of the example shows output from the *Data filtering* stage containing a list of requests and replies in the protocol-independent format. The right part shows a sequence of paired queries with replies, which are the output of the *Data pairing* stage. The system pairs one request and one reply with the special *None* value.

3.2 Model Training

After the *Input Data Processing* stage transformed input PCAP files into a list of request-response pairs, the *Model Training* phase creates a model of the protocol. The model has the form of a finite state machine describing the behavior of the protocol. The system creates the model from provided communication traces. For example, for POP3 protocol, we can consider regular communication traces that represent typical operations, e.g., the client is checking a mail-box on the server, downloading a message from the server or deleting a message on the server. The model is first created for regular communication and later extended with error behavior.

Learning from traces with expected behavior

The model creation process begins by learning the protocol behavior from input data representing regular communication. The result of this training phase is a description of the protocol that represents a subset of correct behavior. The model is created from a collection of individual communication traces. When a new trace is to be added, the tool identifies the longest prefix of the trace that is accepted by the current model. The remaining of the trace is then used to enrich the model. The Figure 4 shows a simple example of creating a model from two correct communication traces (drawn in black ink).

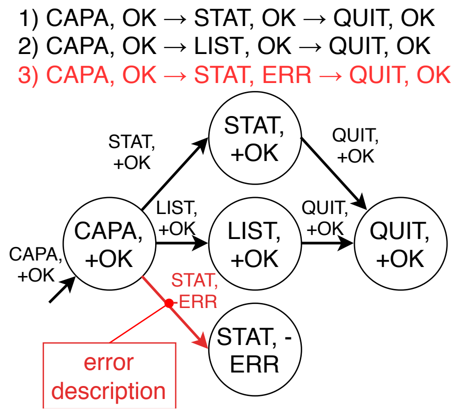

Figure 4: An example of communication traces and the corresponding protocol model. The first two sequences represent correct communication, while the third sequence is communication with an error.

Learning the errors

After the system learns the protocol from regular communication, the model can be extended with error traces. In Figure 4, red arrow stands for a single error transition in the model that corresponds to the added error trace. The system expects that the administrator prepares that error trace as the result of previous (manual) troubleshooting activities. The administrator should also provide error description and information about how to fix the error.

When extending the model with error traces, the procedure is similar to when processing correct traces. Automaton attempts to consume as long prefix of input trace as possible ending in state s. The following cases are possible:

- *Remaining input trace is not empty*: The system creates a new state s' and links it with from state s. It marks the new state as an error state and labels it with a provided error description.
- *Remaining input trace is empty*:
  - State s is error state: The system adds the new error description to existing labeling of an existing state s.
  - State s is correct state: The system marks the state as *possible error* and adds the error description.

When extending the automaton with error traces, it is possible that previously correct state is changed to a possible error state. For consistent application protocols, this ambiguity is usually caused by the abstraction made when describing application protocol behavior.

3.3 Diagnostics

After the system creates a behavioral model that is extended by error states, it is possible to use the model to diagnose unknown communication tracks. The system runs diagnostics by processing a PCAP file in the same way as in the learning process and checks the request/response sequence against the automaton. Diagnostics distinguishes between these classes:

- **Normal:** the automaton accepts the entire input trace and ends in the correct state.
- **Error:** the automaton accepts the entire input trace and ends in the error state.
- **Possible error:** the automaton accepts the entire input trace and ends in the possible error state. In this case, the system cannot distinguish if the communication is correct or not. Therefore, the system reports an error description from the state and leaves the final decision on the user.
- **Unknown:** the automaton does not accept entire the input trace, which may indicate that the trace represents a behavior not fully recognized by the underlying automaton.

If the diagnostic process detects an unknown error or result is not expected, the administrator must manually analyze the PCAP file. After the administrator decides whether the file contains an error or not, the administrator should assign a file to a particular group of files (correct or error) and repeat the learning process. This re-learning process increases the model's ability, and next time the system sees the same situation, it reports the correct result. By gradually expanding, the model covers most of the possible options.

4 ALGORITHMS

This section provides algorithms for (i) creating a model from normal traces, (ii) updating the model from error traces and (iii) evaluating a trace if it contains an error. All three presented algorithms work with a model that uses a deterministic finite automaton (DFA) as its underlying representation.

The protocol behavior is an automaton (Q,Σ,δ,q_0,F). The set of states Q is represented by all query/response pairs identified for the modeled application protocol. As $Q \subseteq \Sigma$, the transition relation $\delta : Q \times \Sigma \rightarrow Q$ is restricted as follows:

$$\delta \subseteq \{((q_s,r_s),(q_i,r_i),(q_i,r_i)) | (q_s,r_s),(q_i,r_i) \in Q\}$$

Each state can be a finite state because the input of the respective input is an indication of the state reached and a list of error descriptions obtained when processing the input data.

4.1 Adding Correct Traces

Algorithm 1 takes the current model (input variable DFA) and adds missing transitions and states based on the input sequence (input variable P). The algorithm starts with the init state and saves it into the *previous_state* variable. The *previous_state* variable is used to create a transition from one state to the next. In each loop of the while loop section, the algorithm assigns the next pair into the *current_state* variable until there is no next pair in the input. From the *previous_state* and the *current_state*, the transition variable is created, and the system checks if the DFA contains this transition. If the DFA does not contain the transition, the transition is added to the DFA. Before continuing with the next loop, the *current_state* variable is assigned to the *previous_state* variable. The updated model will be used as the input for the next unprocessed input sequence. After processing all the input sequences, which represent normal behavior, the resulting automaton is a model of normal behavior.

Algorithm 1 Updating model from the correct traces

```
Inputs: P = sequence of query-reply pairs
        DFA = set of the transitions
Output: DFA = set of the transitions
Previous_state = init_state
while not at end of input P do
    Current_state = get next pair from P
    Transition = Previous_state → Current_state
    if DFA does not contain Transition then
        add Transition to DFA
    Previous_state = Current_state
end
return DFA
```

4.2 Adding Error Traces

The Algorithm 2 has one more input (*Error*), which is a text string describing a user-defined error. The start of the algorithm is the same as in the previous case. The difference is in testing whether the automaton contains the transition specified in the input sequence. If so, the system checks to see if the saved transition also contains errors. In this case, the algorithm updates the error list by adding a new error. Otherwise, the algorithm continues to process the input string to find a suitable place to indicate the error. If the transition does not exist, i is created and marked with the specified error.

Algorithm 2 Extending the model with error traces

```
Inputs: P = sequence of query-reply pairs
        DFA = set of transitions
        Error = description of the error
Output: DFA = set of transitions
Previous_state = init_state
while not at end of input P do
    Current_state = get next pair from P
    Transition = Previous_state → Current_state
    if DFA contains Transition then
        if Transition contains error then
            append Error to Transition in DFA
            return DFA
        else
            Previous_state = Current_state
    else
        add transition Transition to DFA
        mark Transition in DFA with Error
        return DFA
end
return DFA
```

Algorithm 3 Checking an unknown trace

```
Inputs: P = sequence of query-reply pairs
        DFA = set of transitions
Output: Errors = one or more error descriptions
Previous_state = init_state
while not at end of input P do
    Current_state = get next pair from P
    Transition = Previous_state → Current_state
    if DFA contains Transition then
        if Transition contains error then
            return Errors from Transition
        else
            Previous_state = Current_state
    else
        return "unknown error"
end
return "no error detected"
```

4.3 Testing Unknown Trace

The Algorithm 3 uses previously created automaton (DFA variable) to check the input sequence P. According to the input sequence, the algorithm traverses the automaton and collects the errors listed in the transitions taken. If the required transition was not found, the algorithm returns an error. In this case, it is up to the user to analyze the situation and possibly extend the automaton for this input.

5 EVALUATION

We have implemented a proof-of-concept tool which implements the Algorithm 1, 2, and 3 specified in the previous section. In this section, we provide the evaluation of our proof-of-concept tool to demonstrate that the proposed solution is suitable for diagnosing application protocols. Another goal of the evaluation is to show how the created model changes by adding new input data to the model. We have chosen four application protocols with different behavioral patterns for evaluation.

5.1 Reference Set Preparation

Our algorithms create the automata states and transitions based on the sequence of pairs. The implication is that repeating the same input sequence does not modify the learned behavior model. Therefore, it is not important to provide a huge amount of input files (traces) but to provide unique traces (sequences of query-reply pairs). We created our reference datasets by capturing data from the network, removing unrelated communications, and calculating the hash value for each trace to avoid duplicate patterns. Instead of a correlation between the amount of protocols in the network and the amount of saved traces, the amount of files correlates with the complexity of the analyzed protocol. For example, hundreds of DNS query-reply traces captured from the network can be represented by the same sequence (dns_query, dns_reply).

After capturing the communication, all the traces were manually checked and divided into two groups: (i) traces representing normal behavior and (ii) traces containing some error. In case the trace contains an error, we also identified the error and added the corresponding description to the trace. We split both groups of traces (with and without error) into the training set and the testing set.

It is also important to notice that the tool uses these traces to create a model for one specific (or several) network configuration and not for all possible configurations. Focus on a single configuration results in a smaller set of unique traces and smaller created models. This focus allows an administrator to detect situations which may be correct for some network, but it is not correct for a diagnosed network, e.g., missing authentication.

5.2 Model Creation

We have chosen the following four request-reply application protocols with different complexity for evaluation:

Table 2: For each protocol, the amount of total and training traces is shown. These traces are separated into successful (without error) and failed (with error) groups. The training traces are used to create two models, the first without errors and the second with errors. The states and transitions columns indicate the complexity of the created models.

Protocol	**Total traces**		**Training traces**		**Model without error states**		**Model with error states**	
	Successful	**Failed**	**Successful**	**Failed**	**States**	**Transitions**	**States**	**Transitions**
DNS	16	8	10	6	18	28	21	34
SMTP	8	4	6	3	11	18	14	21
POP	24	9	18	7	16	44	19	49
FTP	106	20	88	14	33	126	39	137

- **DNS**: Simple stateless protocol with simple communication pattern - domain name query (type A, AAAA, MX, ...) and reply (no error, no such name, ...).
- **SMTP**: Simple state protocol in which the client has to authenticate, specify email sender and recipients, and transfer the email message. The protocol has a large predefined set of reply codes resulting in many possible states in DFA created by Algorithm 1 and 2.
- **POP**: In comparison with SMTP, from one point of view, the protocol is more complicated because it allows clients to do more actions with email messages (e.g., download, delete). However, the POP protocol replies only with two possible replies (+OK, -ERR), which reduce the number of possible states.
- **FTP**: It is stateful protocol usually requires authentication, then allows the client to do multiple actions with files and directories, and also the protocol defines many reply codes.

The proof-of-concept tool took input data of selected application protocols and created models of the behavior without errors and a model with errors. The Table 2 shows the distribution of the input data into a group of correct training traces and a group of traces with errors. Remaining traces will be later used for testing the model. The right part of the table shows the complexity of the generated models in the format of states and transitions count.

Based on the statistics of models, we have made the following conclusions:

- transitions count represents the complexity of the model better than the state's count;
- there is no direct correlation between the complexity of the protocol and the complexity of the learned model. As can be seen with protocols DNS and SMTP, even though the model SMTP is more complicated than DNS protocol, there were about 50% fewer unique traces resulting in a model with 21 transitions, while the DNS model consists of 34 transitions. The reason for this situation is that one DNS connection can contain more than one query-reply and because the protocol is stateless, any query-reply can follow the previous query-reply value.

Figure 5 shows four charts of four protocols that show the complexity of the models in terms of the number of states, transitions, and testing traces with error results. Each chart consists of two parts. The first part marked as *training from correct traces* creates the model only from traces without errors. To check the correctness of the model, we used testing traces with and without errors. The second part *learning the errors* takes the model created from all the successful traces and extends it with training traces with known errors. To mark the testing trace without error as a correct result, the model has to return that the trace is without error. Testing traces with an error are marked as correct when an error was detected, and the model found the correct error description.

The charts in Figure 5 shows the progress of changing the model size when new traces are added to the model. We have created these values from 25 tests, and the charts show a range of the values from these tests with their median. Each test began by randomizing the order of the trace files, resulting in a different trace order in each test. Based on the deviation of values from the median, we can see that during the learning process, the model is dependent on the order of the traces. However, after we have added all the traces, the created model has the same amount of states and transitions (zero deviation). The zero deviation can be seen at the end of training from correct trace states and also at the end of learning the error states. We have also used a diff tool to compare the final models between themselves to confirm that all the models were the same, and the final model does not depend on the order of the input traces.

Figure 5 shows that by adding new traces, the size of the model is increasing. With the increasing size of the model, the model is more accurate, and

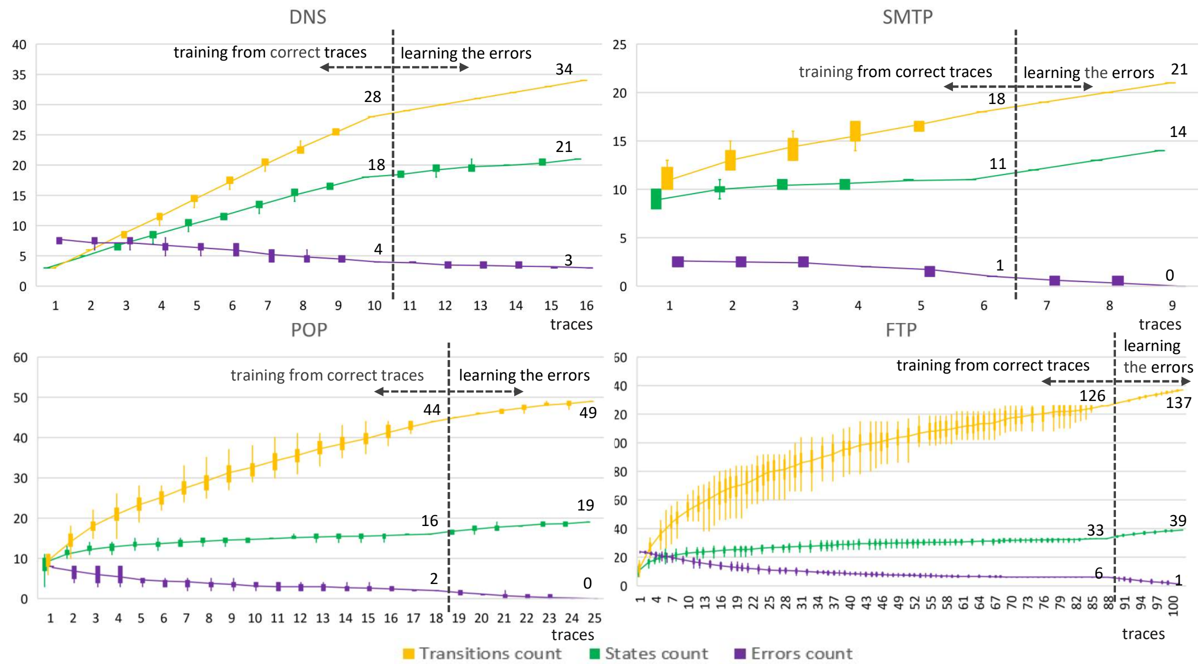

Figure 5: The figure shows the count of transitions, states, and errors in the four analyzed protocols. An error is an incorrect diagnostic result. The values are extracted from 25 random tests, and the median of their values is represented by interconnection lines. The learning process is split into two parts: i) training the model only from traces without any error and after all correct traces have been learned, in section ii) model is extended with the knowledge of known errors.

the amount of diagnostic error results decreases. However, after some amount of traces, the model expansion will slow down until it stops after the tool has observed all valid traces. Stopping the expansion may seem like the point when the model is fully trained, however from our experience, it is not possible to determine when the model is fully learned or at least learned from X%. Even if the model does not grow for a long time, it can suddenly expand by processing a new trace (new extensions, programs with specific behavior, program updates).

Another way of specifying how much percent the model is trained is by calculating all possible transitions. The calculation is $(requests\_count * replies\_count)^2$. Of course, many combinations of requests and replies would not make any sense, but the algorithm can never be sure which combinations are valid and which are not. The problem with counting all possible combinations is that without predefined knowledge of diagnosed protocol the tool can never be sure if all possible requests and replies (no matter the combinations) have already be seen or not.

5.3 Evaluation of Test Traces

Table 3 shows the amount of successful and failed testing traces; the right part of Table 3 shows testing results for these data. All tests check whether:

1. a successful trace is marked as correct (TN);
2. a failed trace is detected as an error trace with correct error description (TP);
3. a failed trace is marked as correct (FN);
4. a successful trace is detected as an error or failed trace is detected as an error but with an incorrect error description (FP);
5. true/false (T/F) ratios which are calculated as $(TN+TP)/(FN+FP)$. T/F ratios represents how many traces the model diagnosed correctly.

As the columns T/F ratio in Table 3 shows, most of the testing data was diagnosed correctly. We have analyzed the incorrect results and made the following conclusions:

- **DNS**: False positive - One application has made a connection with the DNS server and keeps the connection up for a long time. Over time several queries were transferred. Even though the model contains these queries, the order in which they came is new to the model. The model returned an error result even when the communication ended correctly. An incomplete model causes this misbehavior. To correctly diagnose all query combinations, the model has to be created from more unique training traces.
- **DNS**: False positive - The model received a new SOA update query. Even if the communication

Table 3: The created models have been tested by using testing traces, which are split into successful (without error) and failed (with error) groups. The correct results are shown in the true negative and true positive columns. The columns false positive and false negative on the other side contain the number of wrong test results. The ratio of correct results is calculated as a true/false ratio. This ratio represents how many testing traces were diagnosed correctly.

Protocol	**Testing traces**		**Testing against model without error states**					**Testing against model with error states**				
	Successful	**Failed**	**TN**	**TP**	**FN**	**FP**	**T/F ratio**	**TN**	**TP**	**FN**	**FP**	**T/F ratio**
DNS	6	2	4	2	0	2	75 %	4	1	1	2	63 %
SMTP	2	1	2	1	0	0	100 %	2	1	0	0	100 %
POP	6	2	6	2	0	0	100 %	6	2	0	0	100 %
FTP	18	6	18	6	0	0	100 %	18	5	1	0	96 %

TN - true negative, TP - true positive, FN - false negative, FP - false positive, T/F ratio - true/false ratio

did not contain the error by itself, it is an indication of a possible anomaly in the network. Therefore, we consider this as the expected behavior.

- **DNS**: False negative - The situation was the same as with the first DNS False positive mistake - the order of packets was unexpected. Unexpected order resulted in an unknown error instead of an already learned error.
- **FTP**: False negative - The client sent a PASS command before the USER command. This resulted in an unexpected order of commands, and the model detected an unknown error. We are not sure how this situation has happened, but because it is nonstandard behavior, we are interpreting this as an anomaly. Hence, the proof-of-concept tool provided the expected outcome.

All the incorrect results are related to the incomplete model. In the real application, it is almost impossible to create a complete model even with many input data. In the stateless protocols (like DNS), it is necessary to capture traces with all combinations of query-reply states. For example, if the protocol defines 10 types of queries, 3 types of replies, the total amount of possible transitions is $(10 * 3)^2 = 900$. Another challenge is a protocol which defines many error reply codes. To create a complete model, all error codes in all possible states need to be learned from the traces.

We have created the tested tool as a prototype in Python language. We have not aimed at testing the performance, but to get at least an idea of how usable our solution is, we gathered basic time statistics. The processing time of converting one PCAP file (one trace) into a sequence of query-replies and adding it to the model took on average 0.4s. This time had only small deviations because most of the time took initialization of the TShark. The total amount of time required to learn a model depends on the amount of PCAPs. The average time required to create a model from 100 PCAPs was 30 seconds.

6 CONCLUSIONS

This paper suggested a method for automatic error diagnostics in network application protocols by creating models for these applications. There are two use-cases for when administrators should use this approach: (i) if an administrator is experienced, the administrator can learn the model to speed-up the diagnostic process; (ii) if an administrator is inexperienced, the administrator can use the model created by an experienced administrator to diagnose the network.

The already existing diagnostic solutions do not have any automation capabilities, require an administrator to create rules describing the normal and error states or the automatically created protocol models are not used for diagnostic purposes.

Our method uses network traces prepared by administrators to create a model representing protocol behavior. The administrator has to separate the correct from the error traces and annotate the error ones. The model is created based on the query-response sequences extracted from the analyzed protocol. For this reason, the model is applicable only for a protocol with a query-reply pattern. The model represents the correct and incorrect protocol behavior, which is used for unknown network trace diagnostics.

The main benefit of having an own trained model is that the model will represent the protocol in a specific configuration and will not accept situations which may be valid only for other networks. The administrator can use the model to speed up the work by automating unknown communication diagnostics. Our solution uses the well-known tool TShark, which supports many network protocols and allows us to use the Wireshark display language to mark the data.

We have implemented the proposed method as a proof-of-concept tool [4] to demonstrate its capabilities. The tool has been tested on four application protocols that exhibit different behavior. Experiments

[4]https://github.com/marhoSVK/semiauto-diagnostics

have shown that it is not easy to determine when a model is sufficiently taught. Even knowing the protocol complexity is not a reliable indication. Because even the simplest protocol we tested needed a larger model than a more complex protocol. Although the model seldom covers all possible situations, it is useful for administrators to diagnose repetitive and typical protocol behavior and find possible errors.

Future work will focus on: (i) finding other automation cases for the created protocol model ; (ii) designing and implementing additional communication modeling algorithms to support other useful communication features ; (iii) a study on possibility of combining different models from multiple algorithms into one complex model; (iv) integrating timing information into DFA edges.

ACKNOWLEDGEMENTS

This work was supported by project "Network Diagnostics from Intercepted Communication" (2017-2019), no. TH02010186, funded by the Technological Agency of the Czech Republic and by BUT project "ICT Tools, Methods and Technologies for Smart Cities" (2017-2019), no. FIT-S-17-3964.

REFERENCES

Aggarwal, B., Bhagwan, R., Das, T., Eswaran, S., Padmanabhan, V. N., and Voelker, G. M. (2009). NetPrints: Diagnosing home network misconfigurations using shared knowledge. *Proceedings of the 6th USENIX symposium on Networked systems design and implementation*, Di(July):349–364.

Anand, A. and Akella, A. (2010). {NetReplay}: a new network primitive. *ACM SIGMETRICS Performance Evaluation Review*.

Antunes, J., Neves, N., and Verissimo, P. (2011). Reverx: Reverse engineering of protocols. Technical Report 2011-01, Department of Informatics, School of Sciences, University of Lisbon.

Burschka, S. and Dupasquier, B. (2017). Tranalyzer: Versatile high performance network traffic analyser. In *2016 IEEE Symposium Series on Computational Intelligence, SSCI 2016*.

Casas, P., Zseby, T., and Mellia, M. (2016). Big-DAMA: Big Data Analytics for Network Traffic Monitoring and Analysis. *Proceedings of the 2016 Workshop on Fostering Latin-American Research in Data Communication Networks (ACM LANCOMM'16)*.

Chen, M., Zheng, A., Lloyd, J., Jordan, M., and Brewer, E. (2004). Failure diagnosis using decision trees. *International Conference on Autonomic Computing, 2004. Proceedings.*, pages 36–43.

Dhamdhere, A., Teixeira, R., Dovrolis, C., and Diot, C. (2007). NetDiagnoser: Troubleshooting network unreachabilities using end-to-end probes and routing data. *Proceedings of the 2007 ACM CoNEXT*.

El Sheikh, A. Y. (2018). Evaluation of the capabilities of wireshark as network intrusion system. *Journal of Global Research in Computer Science*, 9(8):01–08.

Golden, E. and Coffey, J. W. (2015). A tool to automate generation of wireshark dissectors for a proprietary communication protocol. *The 6th International Conference on Complexity, Informatics and Cybernetics, IMCIC 2015*.

Leaden, S. (2007). The Art Of VOIP Troubleshooting. *Business Communications Review*, 37(2):40–44.

łgorzata Steinder, M. and Sethi, A. S. (2004). A survey of fault localization techniques in computer networks. *Science of computer programming*, 53(2):165–194.

Lodi, G., Buttyon, L., and Holczer, T. (2018). Message Format and Field Semantics Inference for Binary Protocols Using Recorded Network Traffic. In *2018 26th International Conference on Software, Telecommunications and Computer Networks, SoftCOM 2018*.

Luo, C., Sun, J., and Xiong, H. (2007). Monitoring and troubleshooting in operational IP-TV system. *IEEE Transactions on Broadcasting*, 53(3):711–718.

Ming Luo, Danhong Zhang, G. P. L. C. (2011). An interactive rule based event management system for effective equipment troubleshooting. *Proceedings of the IEEE Conference on Decision and Control*, 8(3):2329–2334.

Orzach, Y. (2013). *Network Analysis Using Wireshark Cookbook*. Packt Publishing Ltd.

Procházka, M., Macko, D., and Jelemenská, K. (2017). IP Networks Diagnostic Communication Generator. In *Emerging eLearning Technologies and Applications (ICETA)*, pages 1–6.

Samhat, A., Skehill, R., and Altman, Z. (2007). Automated troubleshooting in WLAN networks. In *2007 16th IST Mobile and Wireless Communications Summit*.

Shiva Shankar, J. and Malathi Latha, M. (2007). Troubleshooting SIP environments. In *10th IFIP/IEEE International Symposium on Integrated Network Management 2007, IM '07*.

Tong, V., Tran, H. A., Souihi, S., and Mellouk, A. (2018). Network troubleshooting: Survey, Taxonomy and Challenges. *2018 International Conference on Smart Communications in Network Technologies, SaCoNeT 2018*, pages 165–170.

Traverso, S., Tego, E., Kowallik, E., Raffaglio, S., Fregosi, A., Mellia, M., and Matera, F. (2014). Exploiting hybrid measurements for network troubleshooting. In *2014 16th International Telecommunications Network Strategy and Planning Symposium, Networks 2014*.

Xiao, M. M., Yu, S. Z., and Wang, Y. (2009). Automatic network protocol automaton extraction. In *NSS 2009 - Network and System Security*.

Zeng, H., Kazemian, P., Varghese, G., and McKeown, N. (2012). A survey on network troubleshooting. *Technical Report Stanford/TR12-HPNG-061012, Stanford University, Tech. Rep.*