Application Error Detection in Networks by Protocol Behavior Model

Martin Holkovič[1], Libor Polčák[2][0000−0001−9177−3073], and Ondřej Ryšavý[2][0000−0001−9652−6418]

[1] Brno University of Technology, Faculty of Information Technology, NES@FIT, Bozetechova 1/2, 612 66 Brno, Czech Republic

[2] Brno University of Technology, Faculty of Information Technology, Centre of Excellence IT4Innovations, Bozetechova 1/2, 612 66 Brno, Czech Republic

{iholkovic,polcak,rysavy}@fit.vutbr.cz

Abstract. The identification of causes of errors in network systems is difficult due to their inherent complexity. Network administrators usually rely on available information sources to analyze the current situation and identify possible problems. Even though they are able to identify the symptoms seen in the past and thus can apply their experience gathered from the solved cases the time needed to identify and correct the errors is considerable. The automation of the troubleshooting process is a way to reduce the time spent on individual cases. In this paper, the model that can be used to automate the diagnostic process of network communication is presented. The model is based on building the finite automaton to describe protocol behavior in various situations. The unknown communication is checked against the model to identify error states and associated descriptions of causes. The tool prototype was implemented in order to demonstrate the proposed method via a set of experiments.

Keywords: Network diagnostics · automatic diagnostics · timed automata · protocol model from traces · encrypted data diagnostics · application behavior model.

1 Introduction

Computer networks are complex systems equipped with different network devices and hosts that provide and consume application services. Various types of errors, such as misconfiguration, device failures, network application crashes, or even user misbehavior can cause that expected network functions are not available. Users perceive network problems by the inaccessibility of web services, the degraded performance of network applications, etc. Usually, it is the role of network administrators to identify the cause of problems and to apply corrective activities in order to restore the network functions again.

The network troubleshooting process is often described as a systematic approach to identify, diagnose and resolve problems and issues within a computer network. Despite the published procedures, methods and techniques, and tool support, the network diagnostics is a largely manual and time-consuming process. Troubleshooting often requires expert technical knowledge of network technologies, communication protocols, and network applications. Another complication is that the administrator often needs to check the number of possible sources to find the real source of the problem. It amounts to check log files in network devices or network applications, the current content of various tables, traces of network communication, etc. Although an experienced administrator usually has advanced skills in network troubleshooting that helps the administrator to quickly identify problems there may be situations that are hard to solve and not evident until the detailed network communication analysis is carried out.

The need for advanced tools that support network diagnostics is expressed by most network professionals surveyed in the report presented by Zeng et al. [31]. Existing tools can provide various information about the network, such as service status and performance characteristics, which is useful for problem detection but they often do not provide enough information for the cause identification. In computer networks, there can happen a lot of different problems. Many of them can be identified by using a network traffic analyzer. The traffic analyzer is a software to intercept the data packet flow that in the hand of an experienced administrator enables to check for the latency issues and other networking problems which help to reveal the root cause. However, using a traffic analyzer requires an understanding of different communication protocols. Also, the number of flows that need to be analyzed can be large making the analysis long and tedious task.

In order to improve the network troubleshooting process, we propose to develop a tool that automatically generates a protocol behavior model from the provided examples (traces) of the protocol conversations. In particular, a network administrator is required to provide two groups of files. The first group contains traces of normal (expected) behavior, while the second group consists of known, previously identified error traces. Based on these distinct groups, the tool is able to construct a protocol model that can be later 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.

When designing the system, we assumed some practical considerations:

- It should not need to be required to implement custom application protocol dissectors to understand the communication.
- Application error diagnostics cannot be affected by lower protocols, e.g., version of IP protocol, data tunneling protocol.
- The The model should be easily interpretable and also useful for other activities too, e.g., security analysis, performance analysis.

The main benefit of this work is a new automatic diagnostic method for the detection of errors observable from the application protocol communication. The method is based on the construction of a protocol behavior model that contains both correct and error communication patterns. An 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 an extended version of the paper "*Using Network Traces to Generate Models for Automatic Network Application Protocols Diagnostics*" [13]. We have added and improved several parts that extend the original paper, in particular: i) models of protocols are better described, ii) the processing of ungeneralizable requests (and states) have been completely reworked, iii) the diagnostics engine can now include time information, iv) and preliminary evaluation of encrypted traffic analysis was realized.

The focus of the previous contribution was only on detecting application layer errors in enterprise networks. We did not consider errors occurred on other layers and domains, e.g., wireless communication [24], routing errors [10], or performance issues [19]. However, in this extended version, we are also able to cope with performance problems. Because we are focusing on enterprise networks, we have made some assumptions on the accessiblity of data sources. For instance, we expect that administrators using this approach have full access to network traffic in the network. Even if the communication outside the company's network is encrypted, the traffic between the company's servers and inside the network can be sometimes available unencrypted, or the data can be decrypted by providing server's private key or logging the session keys [3]. However, because the encrypted traffic forms the majority of all communication on the Internet, we also preliminary evaluated whether the presented approach (generating models from traces) is applicable to encrypted traffic.

The paper is organized as follows: Section 2 describes existing work comparable to the presented approach. Section 3 defines model used for diagnostics. Section 4 overviews the system architecture. Section 5 provides details on the method, including algorithms used to create and use a protocol model. Section 6 presents the evaluation of the tool implementing the proposed system. Section 7 discusses some problems related to our approach. Finally, Section 8 summarizes the paper and identifies possible future work.

2 Related Work

Traditionally, error detection in network systems was mostly a manual process performed by network administrators as a reaction to the user reported or detected service unavailability or connectivity loss. As it is a tedious task various tools and automated methods were developed. A survey by [26] classifies the errors to network systems as either *application-related* or *network-related* problems. The most popular tool for manual network traffic analysis and troubleshooting is Wireshark [20]. It is equipped with a rich set of protocol dissectors that enables to view details on the communication at different network layers. However, 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 [12].

Network troubleshooting can be done using active, passive, or hybrid methods [27]. Active methods rely on the tools that generate probing packets to locate

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

network issues [2]. Specialized tools using generated diagnostic communication were also developed for testing network devices [23]. The advantage of active methods is that it is possible to detect a certain class of errors quickly and precisely identify the problem. On the other hand, generating diagnostic traffic may be unwanted in some situations. Passive detection methods rely on information that can be observed in network traffic or obtained from log files, dumps, etc.

During the course of research on passive network diagnostic methods, several approaches were proposed utilizing a variety of techniques. In the rest of this section, we present the different existing approaches to network diagnostics closely that relates to the presented contribution.

Rule-based Methods. Rule-based systems represent the application of artificial intelligence (reasoning) to the problem of system diagnosis. While this approach was mainly popular for automated fault detection of industrial systems, some authors applied this principle to develop network troubleshooting systems. [15] introduced rule-based reasoning (RBR) expert system for network fault and security diagnosis. The system uses a set of agents that provide facts to the diagnostics engine. [9] proposed distributed multi-agent architecture for network management. The implemented logical inference system enables automated isolation, diagnosis, and repairing network anomalies through the use of agents running on network devices. [11] employed assumption-based argumentation to create an open framework of the diagnosis procedures able to identify the typical errors in home networks. 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 [25].

Protocol Analysis. Automatic protocol analysis 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 [30]. Tool *ReverX* [3] 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 [17]. The automated inference of protocol specification (message format or even protocol behavior model) from traffic samples was considered by several authors. [8] presented *Discover*, a tool for automatic protocol reverse engineering of protocol message formats from network traces. The tool works for both binary and text protocols providing accuracy about 90%. The different approach to solve a similar goal was proposed by [29]. They instrumented network applications to observe the operation of processing network messages. Based on this information their method is able to recreate a message format, which is used to generate protocol parser. This work was extended by the same authors in [7] with an algorithm for extracting the state machine for the analyzed protocol. [16] developed a method based on Markov models called *PRISMA*, which infers a functional state machine and message format of a protocol from network traffic alone. While focused on malware analysis, the tool is capable to identify communication behavior of arbitrary services using binary or textual protocols. Generating application-level specification from network traffic is addressed by [28]. They developed a system called *Veritas* that using the statistical analysis on the protocol formats is able to generate a probabilistic protocol state machine to represent the protocol flows.

Statistical and Machine Learning methods. Statistical and machine learning methods were considered for troubleshooting misconfigurations in the home networks by [1] and diagnosis of failures in the large networks by [6]. *Tranalyzer* [4] is a flow-based traffic analyzer that performs traffic mining and statistical analysis enabling troubleshooting and anomaly detection for large-scale networks. *Big-DAMA* [5] is a framework for scalable online and offline data mining and machine learning supposed to monitor and characterize extremely large network traffic datasets.

Automata-based Analysis. Timed automaton is one of the natural representations of the behavior models for communication protocols. For example, [14] uses timed automata to model parallel systems and to detect errors by verifying the satisfaction of given properties. However, they do not assume to learn the model automatically. Another work [21] proposes a heuristic state-merging algorithm that learns the model automatically. They are using NetFlow records and time windows to create models that are later used to detect malware and infected hosts. [18] uses a model described by timed automata to diagnose errors. The system monitors several sensors which values are converted into timed sequences to be accepted by the timed automata, which are able to detect violations of the measured values to the predefined model.

3 Model representation

Diagnosed protocols are described using models that define the protocols' communications as pair sequences. Each pair consists of a request and a reply message, as shown in Figure 1. These requests and replies are pre-specified message types specific for each protocol. In addition to the original paper, models will take the form of a timed finite automaton, which, in addition to the message order, will also contain timestamp - time since the last reply was received. The finite automaton will process the input sequence and will traverse through the model states. The result of the traverse process will be the result of diagnostics.

Each model processes a message sequence that is distinguished by a 6-tuple: source and destination IP address, source and destination port, L4 protocol, and session ID. The session ID is an optional parameter specified for each protocol to distinguish multiple conversations that are transmitted within a single connection. When transferring multiple conversations over a single connection, the model does not describe the entire connection, but only individual conversations.

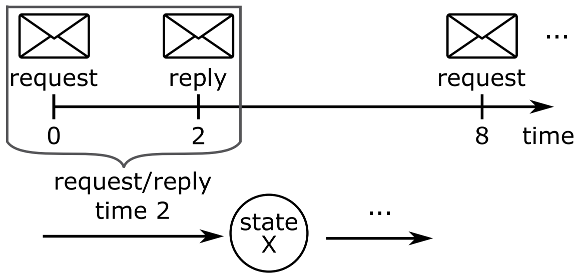

Fig. 1. An illustration of a protocol messages conversion into a finite state automaton. Requests and replies are paired together with the time of their arrival since the last pair.

The finite automaton works with the input alphabet, which is a pair of request and reply values. Both the request and the reply values are composed of packet fields, such as the value of the "*ftp.request.command*" attribute for the FTP request and value of the "*ftp.response.code*" attribute for its reply. If the input symbol (request/reply pair) is repeated (which might mean periodic reports), the model will contain a transition to the same state.

For each request/reply pair, the time since the last reply message or the beginning of the communication is calculated. Using this time, the interval at which the message must arrive for the finite automaton to transition through the state is calculated. Before calculating the interval range, it is necessary to calculate a minimum, a maximum, and a square root of standard deviation from the time values. The interval is calculated in the range from "$minimum - \sqrt{std\ deviation}$" to "$maximum + \sqrt{std\ deviation}$", including extreme values. If the number of values is less than 5, the interval is from zero to infinity (all pairs will match this interval).

A model has a form of a timed finite automaton [22, Def.6.4] — a 6-tuple $(S, S_0, \Sigma, \Lambda, C, \delta)$, where:

- S is a finite set of states,
- $S_0 \in S$ is an initial state,
- Σ is a finite input alphabet ($\Sigma \cap S = \emptyset$, $\epsilon \notin \Sigma$), where $\Sigma =$ (request, reply) and ϵ is an empty value,
- Λ is a finite output alphabet ($\Lambda \cap S = \emptyset$, $\epsilon \notin \Lambda$), where $\Lambda =$ error description and ϵ is an empty value,
- C is a finite state of clocks,
- $\delta \colon S \times (\Sigma \cup \epsilon) \times \Phi(C) \rightarrow S \times \Lambda^* \times 2^C$ is a transition function mapping a triplet of a state, an input symbol (or empty string), and a clock constraint over C to a triplet of a new state, an output sequence, and a set of clocks to be reset. It means, given a specific input symbol, δ shifts the timed transducer from one state to another while it produces an output if and only if the specified clock constraint hold.

4 System Architecture

This section describes the architecture of the proposed system which learns from communication examples and diagnoses unknown communications. In this extended version, the architecture now works with timed information inside automata's transitions, and a new concept of model generalization is described. 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 2, 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.

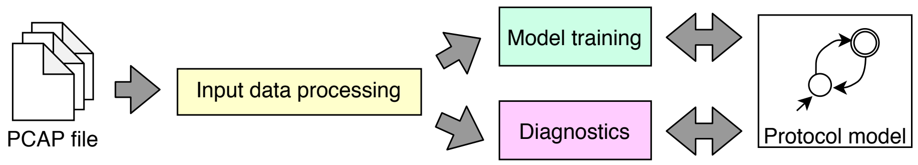

Fig. 2. After the system processes the input PCAP files (the first yellow stage), it uses the data to create the protocol behavior model (the second green stage) or to diagnose an unknown protocol communication using the created protocol model (the-third purple stage). [13]

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

4.1 Input Data Processing

This stage works directly with PCAP files provided by the administrator. Each file is parsed by *TShark* [4] 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.

We have improved the data pairing process in this extended paper to support timed transitions in the model. The system calculates time between the arrival time of the current and the last reply message. For the first reply message within the communication, the time since the beginning of the communication is used. In the case requests do not have corresponding replies, the system uses the requests arrival times. The result of the pairing process is a sequence of pairs with time information, where each pair consists of one request and one reply. The Figure 3 shows an example of this pairing process.

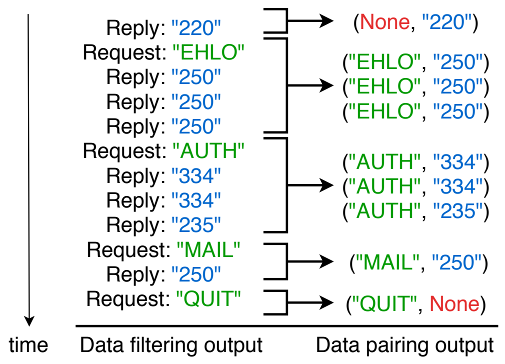

Fig. 3. An SMTP communication in which the client authenticates, sends an email and quits. The left part of the example shows a list of requests and replies together with the time of their arrival in the protocol-independent format. The right part shows a sequence of paired queries with replies, which are the output of the *Input Data Porcessing* stage. For each pair, time since the last pair is also saved. The system pairs one request and one reply with the special *None* value.

4.2 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 protocol model. For example, we can consider regular communication traces that represent typical POP3 protocol operations with the server: the client is checking a mail-box, downloading a message or deleting a message. The model is first created for regular communication and later extended with error behavior.

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

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.

During a traverse within the model, the time attribute of each request-reply pair is added to transitions (each transition has an auxiliary variable containing a list of time attributes). If the number of saved time values within a transition is greater than 5, the time interval of the model transition is recalculated as described in Section 3.

Model generalization. Unfortunately, *TShark* marks some unpredictable data (e.g., authentication data) in some protocols as regular requests and does not clearly distinguish between them. These values are a problem in later processing because these unpredictable values create ungeneralizable states during the model learning phase. Therefore, all transitions that contain requests with unpredictable values are removed from the model and replaced by new transitions.

An unpredictable request value is a request value which is contained inside only one transition - no matter the previous state, the next state, and the reply value. The wildcard value will replace these request values. The time interval of the transition is kept at value from zero to infinity. Which requests contain unpredictable values is determined during the learning process of the model. During this process, the amount of times a request value is being used (no matter the current automata state) is counted (count 1 = unpredictable).

Multiple transitions with unpredictable requests and an identical reply value may originate from a single finite automata state. In this case, all these transitions with the next finite automata states are merged. The merging idea is displayed in Figure 4. After all input traces are used for the model to learn, there is a state from which four transitions are originating. The gray dashed lines are transitions that occurred only once within the input. These two transitions contain various request values, but the same reply value ("OK"). After generalizing these three transitions, a new transition containing the request wildcard value and the "OK" reply value will be added to the model.

When traversing through an automaton, in each state, transitions with explicit commands are checked as first. When no match is found, the model checks if there is a wildcard command value and a reply value for the current state.

Learning the errors. After the system learns the protocol from regular communication, the model can be extended with error traces. 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.

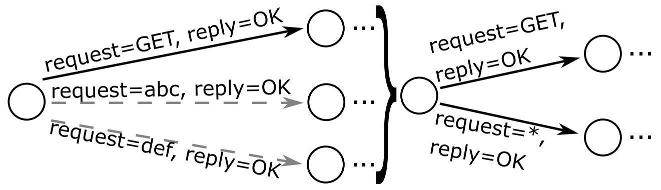

Fig. 4. Illustration of replacing unpredictable requests by a wildcard value (*). The replaced transitions are merged into one generic transition.

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.

4.3 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-reply sequence with their time attributes against the automaton. Diagnostics distinguishes between these classes:

- **Normal:** The automaton accepts the input trace and ends in the correct state.
- **Error:** The automaton accepts the input trace and ends in the error state.
- **Possible error:** The automaton accepts the 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.

It is important to notice that during the traverse within the automaton, the time attribute of each request-reply pair is compared with time constraints. In case the time attribute does not fulfill the constraint, the model generates a warning message. However, the diagnostic process does not stop, and the traverse process continues to the next state in the same way as if the time constraint was fulfilled.

5 ALGORITHMS

This section provides algorithms for (i) creating a model from normal traces, (ii) generalization of the model, (iii) updating the model from error traces and (iv) evaluating a trace if it contains an error. The algorithms are based on algorithms from the original paper. The difference is that in this version, they need to work with timed transitions. All presented algorithms work with a model that uses a deterministic timed finite automaton (DTFA) as its representation.

To simplify algorithms' codes, we have defined time interval $\langle 0; \infty \rangle$ as the default interval. If the interval is not specified, the model uses this value which has less priority when traversing throw the model states. Only when there is no match with a specific interval, the system checks default values.

5.1 Adding Correct Traces

Algorithm 1 takes the input model (DTFA) and adds missing transitions and states based on the input sequence (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 section, the algorithm assigns the next pair into the *current_state* variable until there is no next pair. From the *previous_state* and the *current_state*, the *transition* variable is created, and the system checks if the DTFA contains this *transition*. If the DTFA does not contain it, it is added together with the *time_value*. Otherwise, the new *time_value* is added to the *transition*. If at least five time values are saved, the *time_interval* is calculated and applied to the *transition*.

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 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 = query-reply pairs sequence with time value; DTFA = set of the transitions
Output: DTFA = 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 DTFA does not contain transition then
        add transition to DTFA and save time_value to the transition
    else
        add time_value to the saved times in transition
        if saved times >= 5 then
            calculate the time_interval constraint and apply it to the transition
    previous_state = current_state
end
return DTFA
```

Algorithm 2 Generalization of the model

```
Inputs: DTFA = set of transitions
Output: DTFA = set of transitions
foreach $transition \in DTFA$ do
    if transition contains only one time then
        new_transition = make copy of transition
        remove transition from $DTFA$
        replace request in new_transition by wildcard
        if $DTFA$ does not contain new_transition then
            add new_transition to DTFA
end
return DTFA
```

5.2 Model Generalization

The Algorithm 2 takes all transitions from a model one by one (variable *transition*), calculates the number of times each *transition* was used, and checks whether the *transition* was used only once (contains only one time value). Only one time value means that in all of the input traces, the *transition* was used only once. The model creates a new copy of the *transition* (variable *new_transition*) and removes the old one.

The wildcard value replaces the request value in the *new_transition*. The algorithm checks whether the model contains this *new_transition*, and if not, it is inserted into the model. This presence control ensures that a single transition replaces multiple ungeneralizable states with a wildcard request value.

Algorithm 3 Extending the model with error traces

```
Inputs: P = query-reply pairs sequence; DTFA = set of transitions; Error = description
        of the error
Output: DTFA = 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 $\rightarrow$ current_state
    if $DTFA$ contains transition then
        if transition fulfills time_interval then
            if transition contains error then
                append error to transition in DTFA
                return DTFA
            previous_state = current_state
        else
            add transition to DTFA and mark it with error
            return DTFA
    else
        add transition to DTFA and mark it with error
        return DTFA
end
return DTFA
```

5.3 Adding Error Traces

The Algorithm 3 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 whether the *transition* fulfills the *time_interval*. This time interval checking is an improvement of the algorithm from the previous paper. Only when the *time_interval* is fulfilled, 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 fulfill the *time_interval* restriction or the *transition* does not exist, it is created and marked with the specified *error*.

5.4 Testing Unknown Trace

The Algorithm 4 uses previously created automaton (DTFA variable) to check the input sequence P. According to the input sequence, the algorithm traverses the automaton and checks whether the transitions contain errors. If an error in some transition is found, the system returns an errors description messages (*errors*) to the user. If the *transition* was not found, the algorithm returns an unknown error. In this case, it is up to the user to analyze the situation and possibly extend the automaton for this input.

In this extended paper, the system also verifies if the input sequence fulfills transitions time restrictions. With each *transition*, the time value is compared to the *time_interval*. If the *transition* does not fulfill the *time_interval*, the system creates a warning message to the user. More than one warning message can be generated because the generating of warning messages does not stop the diagnostic process.

Algorithm 4 Checking an unknown trace

```
Inputs: P = query-reply pairs sequence; DTFA = 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 DTFA contains transition then
        if transition doesn't fulfill time_interval then
            create warning that transition does not matched the interval and continue
        if transition contains error then
            return errors from transition
        previous_state = current_state
    else
        return "unknown error"
end
return "no error detected"
```

6 EVALUATION

We have implemented a proof-of-concept tool which implements the Algorithm 1, 2, 3, and 4. 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.

The results from the original's subsections 5.1-5.3 are the same and still valid. From this reason the subsections 6.1 and 6.3 are the same as in the original paper, and in the subsection 6.2 the figure showing the model's complexity during the model training is omitted. The new content is in the following subsections. Section 6.4 tests the benefit of using finite automata as the model by detecting a performance problem inside a communication. The last section 6.5 tries to verify whether the proposed approach is somehow usable for encrypted traffic.

6.1 Reference Set Preparation and Model Creation

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 query-reply sequence ($A\ type\ query, No\ error$).

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 into the training set and the testing set.

It is important to notice that the tool uses traces to create a model for one specific 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 allows an administrator to detect situations which may be correct for some network, but not for a diagnosed network, e.g., missing authentication.

6.2 Model Creation

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

- **DNS**: Simple stateless protocol with 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 DTFA created by Algorithm 1 and 2.
- **POP**: In comparison with SMTP, 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**: Stateful protocol allowing the client to do multiple actions with files and directories on server. The protocol defines many reply codes.

Table 1. For each protocol, the amount of total and training traces is shown. These traces are separated into *proper* (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 show the complexity of the models. [13]

Protocol	**Total traces**		**Training traces**		**Model without error states**		**Model with error states**	
	Proper	**Failed**	**Proper**	**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

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 1 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 sum depicts the model's complexity better than the state's sum;
- there is no direct correlation between the complexity of the protocol and the complexity of the model. As can be seen with protocols DNS and SMTP, even though the model SMTP is more complicated than DNS model, there were about 50% fewer unique traces resulting in a model with 21 transitions, while the DNS model consists of 34 transitions. The reason 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.

Part of the original paper is a figure with four charts outlining the same four protocols, as displayed in Table 1. These four charts show the progress of increasing the model size and decreasing the number of diagnostic errors when new traces are added to the model. The model creation process was split into two parts: training from traces without errors and learning the errors.

6.3 Evaluation of Test Traces

Table 2 shows the amount of successful and failed testing traces; the right part of Table 2 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.

Table 2. The created models have been tested by using testing traces, which are split into *proper* (without error) and *failed* (with error) groups. The correct results are shown in the true negative (*TN*) and true positive (*TP*) columns. The columns false positive (*FP*) and false negative (*FN*) on the other side contain the number of wrong test results. The ratio of correct results is calculated as a true/false ratio (*T-F ratio*) . This ratio represents how many testing traces were diagnosed correctly. [13]

Protocol	**Testing traces**		**Testing against model without error states**					**Testing against model with error states**				
	Proper	**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

As the columns T-F ratio in Table 2 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 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 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. Our goal was not to test 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. At average, to create a model from 100 PCAPs, 30 seconds was required.

6.4 Timed transitions

For this test, we took a model of the SMTP protocol from the previous test, and we have extended it with new PCAP files. These new PCAP files contain two problems that could not be detected without a timed finite automata model:

1. **overloaded SMTP server** - all requests from the server have a high delay;
2. **overloaded authentication LDAP server** - the SMTP server responds to requests at an average speed, but user authentication, which uses an external LDAP server takes considerably longer.

Unfortunately, we do not have PCAP files with these errors from a real production network, so we had to create them. We achieved this by manually overloading the SMTP server, LDAP server, or creating a delay for the communication between these two servers.

The part of the extended model that covers authentication problems is displayed in Figure 5. Red colored transitions cover situations where, regardless of the authentication type and authentication result, a slow response is detected. The model describes two authentication methods: simple authentication (name and password in one message) and login authentication (name and password sent separately). As described in Algorithm 3, these new transitions have a time interval with the value $< 0; \infty >$. Therefore all traces that do not match the original time restrictions are matched by these new transitions.

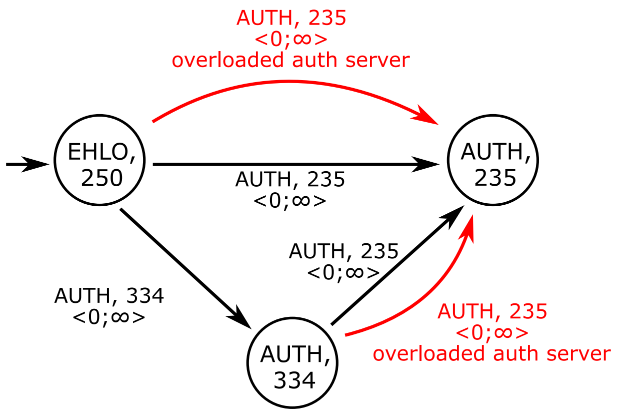

Fig. 5. The segment of the SMTP model which contains new transitions and states related to the high delay from the authentication server.

We have tested the created model on other captured PCAP files. With overloaded LDAP servers or high communication latency between an SMTP and an LDAP server, the model correctly detected a problem related to the authentication. When the SMTP server was overloaded, the model correctly detected overload at the beginning of the communication.

However, the extended SMTP model was not able to correctly diagnose a situation where the beginning of the communication was OK, and the overload of the SMTP server began during client authentication. Although other delayed responses followed the delayed response for authentication, the system stopped at the first error and erroneously detected an authentication problem.

6.5 Encrypted data diagnostics

We have performed another type of evaluation aimed at verifying if the method proposed by us applies to encrypted traffic or not. As described in the previous sections, the diagnostic process uses request-reply values, but we are not able to detect this in encrypted communication. To overcome this limitation, we have proposed a modification to the model in the way that the model uses the size of the encrypted data (TLS record size) instead of the request-reply value.

Because we only consider the size of the application data, which can easily vary even if the request value or the reply value is the same, it is necessary to work with a range of values. We are using an algorithm similar to the one used to calculate the range for time intervals. From the set of values, we calculate the minimum, maximum, and square root of the standard deviation. The range of the interval that will accept messages will have a value ranging from "$minimum - \sqrt{std\ deviation}$" to "$maximum + \sqrt{std\ deviation}$". The difference from the calculation of the time intervals is that with a range of application data sizes, the interval is calculated from even a single value, and it is not required to have at least five values.

With this modified approach of diagnostics, we are not able to diagnose such a range of errors as in unencrypted traffic. However, we are still able to obtain at least basic information about the state of communication. We have based this idea on the fact that protocol communication between endpoints goes through different states. Protocol standards specify these states and their order. Diagnostics of encrypted communication, do not analyze exactly what caused the error but only when (or in which state) the error occurred.

As in the case of unencrypted communications, a model should be created only from traces belonging to one service (on a single server) and applied to the same service. With other configurations, the content of messages can be different, which would cause different sizes of the messages themselves.

To verify the idea of diagnostics based on the size of application data, we have captured ten correct and three error SMTP communication traces. From these communications, a model was created, which is shown in Figure 6. The Figure shows three detectable errors that also separate the SMTP protocol states - welcoming client, user authentication, and e-mail sending. Based on this test, we have reached the conclusion that the approach is usable. However, to use the model in the real-world, the model should be trained from more traces.

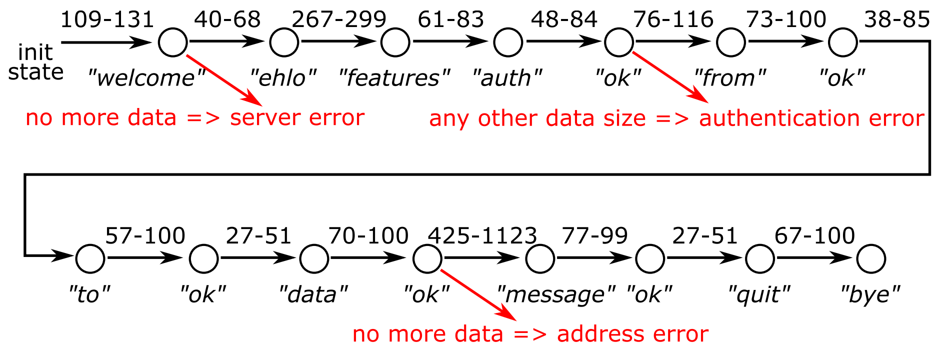

Fig. 6. The segment of the SMTP model which contains new transitions and states related to the high delay from the authentication server.

7 DISCUSSION

This chapter describes some of the topics we have come across when developing and using the tool.

7.1 Fully trained model

One of the fundamental questions when using the tool is when the model is fully (or for X%) trained and when it is possible to switch from training mode to diagnostic mode. The simplest way of specifying how much percent the model is trained is by calculating all possible transitions. Transitions are connecting any two states, which are defined by request/reply values. The total number of states is $requests\_count * replies\_count$, and the total number of transitions is $states\_count^2$. Of course, many combinations of requests and replies do not make 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 the diagnosed protocol, the tool can never be sure if all possible requests and replies have already been seen or not.

One way to determine whether a model is trained without knowing the total number of states is by checking the list of trained states when processing new input data. If the system has not detected a new state for a certain amount of iterations, it will declare that the list is complete, and the model is fully trained. Here comes the problem of determining how long to wait for a new value.

Basically, there are three approaches that can be combined:

A1 amount of new files - waiting for X new files to be processed (e.g., 100);
A2 training duration- waiting for an X lasting interval;
A3 unique amount of clients - waiting for X unique clients (e.g., 10).

Unfortunately, each of these approaches has drawbacks that cannot be eliminated entirely:

ReA1 If most (or all) new files have the same content type (for example, the same client queries the same DNS translation type), then the number of these files is not important. We have partially solved this problem by creating communications fingerprints and ignoring duplicate fingerprint files. This is why the evaluation section describes so few unique communications. As an example, we take the SMTP protocol. Most conversations had an identical pattern - welcoming the client, user authentication, and sending an email. The responses to all commands were without any error. Although the welcome message (timestamp), login information, email addresses, and email content varied from one communication to the next, the fingerprint was the same as all of this data was deleted in the *Input Data Processing* stage. So even though we had dozens of these conversations, we counted them as just one conversation.

ReA2 By taking communications carried during a limited time frame, e.g. 24 hours, we may not cover situations that arise less frequently or irregularly. For example, SMTP clients can process requests to send a message even when the client is offline and then send these messages at once when the client is online again. However, the situation, when a client sends multiple messages at once, does not occur often, and a 24-hour window might not be enough.

ReA3 This criterion can only be applied if a large number of clients connect to a single server. In the case of a pre-defined server-server communication, this criteria makes no sense. Even with a larger amount of clients, all clients may use the same application, the same settings, and perform the same activity.

In our opinion, the best option is to combine all three mentioned approaches and select parameters so that the data sample used is relevant and that the model is trained within an acceptable time. For example, waiting for 100 unique sequences in the SMTP protocol or 1000 communications if only one communication happens per day is meaningless. However, even in this case, we are not able to capture the following situations:

- Protocol updates can introduce new version of the protocol which may introduce new types of commands or responses.
- Another version (e.g., by an update) of the application or a brand new application will appear on the network. This may cause the client to start to communicate with the server with a different pattern of behavior.
- Some types of errors are associated with less frequently used features, which occur very irregularly. Such errors are hard to catch and get into the model.

From our experience, it is not possible to determine when the model is fully trained or at least trained 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).

Nevertheless, the model incorporates means to train even in the diagnostic phase (when the tool is deployed). An administrator that encounters a false error can always improve the model. Consequently, as time passes, the model can adapt to handle infrequent communications and protocol/application updates.

7.2 Data labeling

During model training, an administrator needs to determine if there is an error in conversation manually. If an error is detected, an administrator creates a description of this error. This process is time-consuming and requires knowledge of the modeled protocol and computer networks in general. However, it is important to realize that in the case of manual diagnosis, the administrator has to perform a similar diagnosis. Hence, our approach does not introduce additional requirements for administrators' skills. Therefore, we do not think that the need to manually mark communications is a disadvantage of our method.

Another possible way to label data can be by applying artificial intelligence or machine learning. However, we think that even with machine learning, supervised learning has to be used. Therefore, it is still necessary to analyze the content of the communications manually and instruct the algorithm. Another question is, how easy it is for the network administrator to work with artificial intelligence, and whether network administrators without programming knowledge understand working with machine learning.

7.3 Model of models

As part of our proposed approach, we do not model relationships between individual communications or between different protocols. Each model describes one particular communication with one protocol. However, there are more complex errors that cannot be detected or diagnosed by analyzing just a single communication. An example is downloading a web page content. This activity can consist of multiple individual communications: user authentication, HTML page download, and download of other elements such as images or scripts. Another example is that during communication with an application server, the server establishes another connection to the RADIUS server to authenticate the user.

To be able to diagnose problems that are spread across multiple communications correctly (even over multiple protocols), it is necessary to create a model which will consist of several models describing individual communications ("model of models"). This high-level model can check one communication and, based on its result, launch another model for the following communication or generate a diagnostic report.

7.4 Another usage of models

The proposed models do not apply to network diagnosis only. Another application is the security analysis. The model can be trained to accept only communications which fulfill the security policy. Other communications that are not accepted by the trained model are reported as possibly dangerous. Another type of security analysis is by visualizing the model and employing a manual analysis. Our tool can export models to a format suitable for graphical visualization. From the trained models, an administrator can make some deductions. For example, if some users are not using the recommended authentication or some communications contain outdated commands.

Another possible model usage is related to time transitions within the model. We think it makes sense to investigate whether it is possible to use models for profiling communications. For example, in the case of FTP communication, if the browsing and downloading of files are without delays caused by user interaction, it is possible to associate such communication with a tool that automatically browses and downloads server content.

8 CONCLUSIONS

In the presented paper, we have proposed an automatic method for generating automata from network communication traces and their use in the network diagnostic process. The diagnostic system is designed to learn from both normal error-free communication sequences as well as from erroneous traces in order to create an automata-based model for the communication protocol behavior. The states in the automaton can be labeled with additional information that provides diagnostic information for the error detected.

The method requires network traces prepared by an expert to create a good model. The expert is expected to annotate network traces and label the known errors. The current model is only applicable to query-response protocols and those that provides a sufficient amount of information to observe their state. We demonstrated that if the model is created based on the reasonable sample of good and error behavior it can be used in any network environment.

We have implemented the method in a proof-of-concept tool[5] and use it in a set of experiments for demonstration purposes. The tool has been tested on a limited set of application protocols of different types, e.g., e-mail transfer, file download, domain name resolution. Experiments show that the suitability and usability of the model heavily depend on the network protocol. Although the model typically does not cover all possible scenarios, it is useful for diagnosis of repetitive error. As the model can learn errors during deployment, an administrator does not have to deal with errors not encountered during learning phase more than once.

ACKNOWLEDGEMENTS

This work was supported by The Ministry of Education, Youth and Sports from the National Programme of Sustainability (NPU II) project IT4In-novations excellence in science - LQ1602".

References

1. Aggarwal, B., Bhagwan, R., Das, T., Eswaran, S., Padmanabhan, V.N., Voelker, G.M.: NetPrints: Diagnosing home network misconfigurations using shared knowledge. Proceedings of the 6th USENIX symposium on Networked systems design and implementation **Di**(July), 349–364 (2009), http://portal.acm.org/citation.cfm?id=1559001
2. Anand, A., Akella, A.: Net-replay: a new network primitive. ACM SIGMETRICS Performance Evaluation Review (2010). https://doi.org/10.1145/1710115.1710119
3. Antunes, J., Neves, N., Verissimo, P.: Reverx: Reverse engineering of protocols. Tech. Rep. 2011-01, Department of Informatics, School of Sciences, University of Lisbon (2011), http://hdl.handle.net/10451/14078

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

4. Burschka, S., Dupasquier, B.: Tranalyzer: Versatile high performance network traffic analyser. In: 2016 IEEE Symposium Series on Computational Intelligence, SSCI 2016 (2017). https://doi.org/10.1109/SSCI.2016.7849909
5. Casas, P., Zseby, T., Mellia, M.: 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) (2016). https://doi.org/2940116.2940117
6. Chen, M., Zheng, A., Lloyd, J., Jordan, M., Brewer, E.: Failure diagnosis using decision trees. International Conference on Autonomic Computing, 2004. Proceedings. pp. 36–43 (2004). https://doi.org/10.1109/ICAC.2004.1301345, http://ieeexplore.ieee.org/document/1301345/
7. Comparetti, P.M., Wondracek, G., Krügel, C., Kirda, E.: Prospex: Protocol specification extraction. 2009 30th IEEE Symposium on Security and Privacy pp. 110–125 (2009)
8. Cui, W., Kannan, J., Wang, H.J.: Discoverer: Automatic Protocol Reverse Engineering from Network Traces. USENIX Security (2007). https://doi.org/"Protocol-Independent Adaptive Replay of Application Dialog"
9. De Paola, A., Fiduccia, S., Gaglio, S., Gatani, L., Lo Re, G., Pizzitola, A., Ortolani, M., Storniolo, P., Urso, A.: Rule based reasoning for network management. In: Seventh International Workshop on Computer Architecture for Machine Perception (CAMP'05). pp. 25–30 (July 2005). https://doi.org/10.1109/CAMP.2005.47
10. Dhamdhere, A., Teixeira, R., Dovrolis, C., Diot, C.: NetDiagnoser: Troubleshooting network unreachabilities using end-to-end probes and routing data. Proceedings of the 2007 ACM CoNEXT (2007). https://doi.org/10.1145/1364654.1364677
11. Dong, C., Dulay, N.: Argumentation-based fault diagnosis for home networks. In: Proceedings of the 2nd ACM SIGCOMM Workshop on Home Networks. p. 37–42. HomeNets '11, Association for Computing Machinery, New York, NY, USA (2011). https://doi.org/10.1145/2018567.2018576, https://doi.org/10.1145/2018567.2018576
12. El Sheikh, A.Y.: Evaluation of the capabilities of wireshark as network intrusion system. Journal of Global Research in Computer Science **9**(8), 01–08 (2018)
13. Holkovič, M., Ryšavý, O., Polčák, L.: Using network traces to generate models for automatic network application protocols diagnostics. In: Proceedings of the 16th International Joint Conference on e-Business and Telecommunications Volume 1: DCNET, ICE-B, OPTICS, SIGMAP and WINSYS. pp. 43–53. SciTePress - Science and Technology Publications (2019), https://www.fit.vut.cz/research/publication/12012
14. Ivković, N., Milić, L., Konecki, M.: A timed automata model for systems with gateway-connected controller area networks. In: 2018 IEEE 3rd International Conference on Communication and Information Systems (ICCIS). pp. 97–101. IEEE (2018)
15. Kim, S., Ahn, S.j., Chung, J., Hwang, I., Kim, S., No, M., Sin, S.: A rule based approach to network fault and security diagnosis with agent collaboration. In: Kim, T.G. (ed.) Artificial Intelligence and Simulation. pp. 597–606. Springer Berlin Heidelberg, Berlin, Heidelberg (2005)
16. Krueger, T., Gascon, H., Krämer, N., Rieck, K.: Learning stateful models for network honeypots. In: Proceedings of the 5th ACM Workshop on Security and Artificial Intelligence. p. 37–48. AISec '12, Association for Computing Machinery, New York, NY, USA (2012). https://doi.org/10.1145/2381896.2381904, https://doi.org/10.1145/2381896.2381904

17. Lodi, G., Buttyon, L., Holczer, T.: 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 (2018). https://doi.org/10.23919/SOFTCOM.2018.8555813
18. Lunze, J., Supavatanakul, P.: Diagnosis of discrete–event system described by timed automata. IFAC Proceedings Volumes **35**(1), 77–82 (2002)
19. Ming Luo, Danhong Zhang, G.P.L.C.: An interactive rule based event management system for effective equipment troubleshooting. Proceedings of the IEEE Conference on Decision and Control **8**(3), 2329–2334 (2011). https://doi.org/10.1007/s10489-005-4605-0
20. Orzach, Y.: Network Analysis Using Wireshark Cookbook. Packt Publishing Ltd (2013)
21. Pellegrino, G., Lin, Q., Hammerschmidt, C., Verwer, S.: Learning behavioral fingerprints from netflows using timed automata. In: 2017 IFIP/IEEE Symposium on Integrated Network and Service Management (IM). pp. 308–316. IEEE (2017)
22. Polčák, L.: Lawful Interception: Identity Detection. Ph.d. thesis, Brno University of Technology, Faculty of Information Technology (2017), https://www.fit.vut.cz/study/phd-thesis/679/
23. Procházka, M., Macko, D., Jelemenská, K.: IP Networks Diagnostic Communication Generator. In: Emerging eLearning Technologies and Applications (ICETA). pp. 1–6 (2017)
24. Samhat, A., Skehill, R., Altman, Z.: Automated troubleshooting in WLAN networks. In: 2007 16th IST Mobile and Wireless Communications Summit (2007). https://doi.org/10.1109/ISTMWC.2007.4299084
25. łgorzata Steinder, M., Sethi, A.S.: A survey of fault localization techniques in computer networks. Science of computer programming **53**(2), 165–194 (2004)
26. Tong, V., Tran, H.A., Souihi, S., Mellouk, A.: Network troubleshooting: Survey, Taxonomy and Challenges. 2018 International Conference on Smart Communications in Network Technologies, SaCoNeT 2018 pp. 165–170 (2018). https://doi.org/10.1109/SaCoNeT.2018.8585610
27. Traverso, S., Tego, E., Kowallik, E., Raffaglio, S., Fregosi, A., Mellia, M., Matera, F.: Exploiting hybrid measurements for network troubleshooting. In: 2014 16th International Telecommunications Network Strategy and Planning Symposium, Networks 2014 (2014). https://doi.org/10.1109/NETWKS.2014.6959212
28. Wang, Y., Zhang, Z., Yao, D.D., Qu, B., Guo, L.: Inferring protocol state machine from network traces: A probabilistic approach. In: Lopez, J., Tsudik, G. (eds.) Applied Cryptography and Network Security. pp. 1–18. Springer Berlin Heidelberg, Berlin, Heidelberg (2011)
29. Wondracek, G., Comparetti, P.M., Kruegel, C., Kirda, E.: Automatic network protocol analysis. In: Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS'08 (2008)
30. Xiao, M.M., Yu, S.Z., Wang, Y.: Automatic network protocol automaton extraction. In: NSS 2009 - Network and System Security (2009). https://doi.org/10.1109/NSS.2009.71
31. Zeng, H., Kazemian, P., Varghese, G., McKeown, N.: A survey on network troubleshooting. Technical Report Stanford/TR12-HPNG-061012, Stanford University, Tech. Rep. (2012)