Using Rule-Based Decision Trees for Automatic Passive Diagnostics of the Network Problems

Martin Holkovič

Flowmon Networks
Sochorova 3232/34
Brno 61600, CZ
Email: martin.holkovic@flowmon.com

Ondřej Ryšavý

Faculty of Information Technology
Brno University of Technology
Brno 61266, CZ
Email: rysavy@fit.vutbr.cz

***Abstract*—Network troubleshooting often requires a detailed analysis that may involve network packet capturing and a manual analysis using tools such as Wireshark. This is time-consuming and requires deep knowledge of communication protocols. Therefore, this domain is a suitable candidate for the deployment of an expert system. In this paper, we consider a rule-based system integrating the expert knowledge that performs an automatic root cause analysis of network problems identifiable from network communications. The system is open, thus it is possible to add new rules as needed, e.g., for specific and recurring cases of a target environment. The rules are evaluated in a tree-based fashion, which enables us to collect additional information during the problem search to better explain the possible causes. We successfully deployed the tool as part of a commercial tool for network monitoring.**

Keywords–Network diagnostics; rule-based diagnostics; fault tree analysis; event-based diagnostics; decision trees.

I. Introduction

Network infrastructure and applications are complex, prone to cyber attacks, outages, performance problems, misconfigurations, and problems caused by software or hardware incompatibility. All these problems may affect network performance and user experience [2], which may have fatal consequences for critical network infrastructure, e.g., e-health, e-government, Industrial IoT, smart grid, etc. Network troubleshooting is thus among the most common and important activities by network administrators. Despite the help of the current network monitoring tools, identification of a root cause of issues can be a complicated and mostly manual activity. The tools often reveal symptoms of the problem but the reasoning and problem localization are left for human operators expecting that they understand the problem and have sufficient knowledge of the technologies involved. Even if it is the case, the troubleshooting can be a lengthy and tiresome process that requires inspection of different sources of information, e.g., log files, the content of various tables, communication traces, etc. Application communication protocols are designed to implement the data exchange of remote parties. The protocol specification defines the syntax and meaning of messages, the way the conversation is controlled, and also the indication of error states. Thus, by inspecting the network communication it is possible to understand the situation and identify the indicated errors and in many cases also their probable cause.

Unfortunately, many network administrators do not have the proper tools and/or knowledge to diagnose and fix network problems effectively, and they require an automated tool to diagnose these errors [3]. Zeng et al. [4] provide a short survey on network troubleshooting from the administrators' viewpoint identifying the most common network problems: *reachability problems, degraded throughput, high latency, and intermittent connectivity*. The consulted network administrators expressed the need for a network monitoring tool that would be able to identify such problems.

This paper proposes a system, which creates diagnostic information only by performing passive network traffic packet-level analysis. Previous research and development provided tools for helping administrators to diagnose faults [5] and performance problems [6]. However, these tools either require *installation of agents on hosts, active monitoring, or providing rich information about the environment*. The idea behind our proposal is to automate the reasoning usually done by network analysts when investigating the root cause of an error from the captured network traces. It means that it is not necessary to change the network environment nor deploy any new devices. The troubleshooting process may remain unchanged except that one of the most labor-intensive parts represented by the packet-level traffic analysis is automated. Still, the user can verify the results obtained from the automated analysis as the process provides sufficient diagnostic information for the identification of problem relevant artifacts.

One of the most common ways of analyzing network traffic is by using a network packet analyzer (e.g., Wireshark). The analyzer works with captured network traffic (PCAP files) and displays structured information of layered protocols contained in every packet (encapsulated protocols, protocol fields). Administrators work with this information, check transferred content and compare the data with expected values. This process, done manually, is time-consuming and requires a good knowledge of network protocols and technologies.

The main contribution of this paper is a proposal of a tool for automatic diagnoses of network related problems from network communication only. Our approach tries to imitate a diagnostic process of a real administrator using the fault tree method and a popular packet parsing tool TShark. We have also implemented a proof-of-concept implementation to confirm the viability of the approach. This paper is an extension of our previous paper [1]. The most significant change is the improvement of input data processing. A new more efficient mechanism of converting input data into a specific indexable

format has been implemented. This change required significant modification of the method the system uses to access the data. However, the new format simplifies processing of other data types and reduces the execution time of the whole diagnostic significantly. A simple example of a tool usage for another data type (log files) is also presented.

The paper is organized as follows. Section II defines the problem statement and research questions. Section III discusses related work and describes diagnostic approaches. Our solution consists of five stages and is introduced in Section IV. Section V instructs network administrators how to use our system and shows how we model diagnostic knowledge. Section VI shows the output from the tool and evaluates the performance. Finally, Section VII is the conclusion, which summarizes the current state and proposes future work.

II. Research Questions

Our primary goal is to design a system that infers possible causes accountable for network related problems, such as service unreachability or application errors. Offering a list of actions for fixing the errors' cause is the secondary and optional goal. All this information is gathered only from captured network communication, which makes this approach applicable to various scenarios.

In our work, we focus on enterprise networks that have complex networking topologies, usually consisting of various network and end-point devices. The availability of network traces in the form of packet captures is essential to our method. Thus, we expect that administrators can collect network communication at appropriate locations in the network. Also, we consider that the capturing process creates packet captures without packet losses. As this may be difficult to guarantee for high-speed networks without using specialized hardware, for the diagnostic we usually do not require all communication. Thus, the packet capture can be recorded by applying a suitable filter to reduce the amount of data that needs to be processed.

To achieve our goal, we need to find answers to the following research questions:

1) How to model different network faults in a suitable way for implementation in a diagnostic system? *Reachability, application specific, and device malfunctioning problems* can cause various networking issues. We need to have a unified approach for modeling these problems to identify the symptoms and link them with root causes.
2) What information should be extracted from the captured network communication to identify symptoms of failures? In our case, we can passively access the communication in the monitored network and extract the necessary data to detect possible symptoms. An approach that can efficiently detect the symptoms in terms of precision and performance is needed.
3) How to identify the root cause of the problem, if we have a set of related symptoms? The core part of the diagnostic engine is to apply knowledge gathered from observed symptoms to infer the possible root cause of the problem. The result should provide the information in sufficient detail. For instance, if the authentication during the establishing of the connection fails, then we would like to know this specific information instead of a more general explanation (e.g., unable to establish a connection).
4) What actions can be provided to the administrator to fix the problems? Based on the observed symptoms and the root cause, the system should be able to provide fixing guidelines. These guidelines are supposed to be easy to understand even for an inexperienced administrator.

III. Related Work

A lot of research activities were dedicated to the diagnoses of network faults. Various methods were proposed for different network environments [5], in particular, home networks [7], enterprise networks [8]–[11], data centers [6], backbone and telecommunications networks [12], mobile networks [13], Internet of Things [14], Internet routing [15] and host reachability. Methods of network troubleshooting can be roughly divided into the following classes:

Active methods use traffic generators to send probe packets that can detect the availability of services or check the status of applications [16]. Usually, generators create diagnostic communication according to the test plan [8]. The responses are evaluated and provide diagnostic information that may help to reveal device misconfiguration or transient fail network states. Diagnostic probes introduce extra traffic, which may pose a problem for large installations [11]. Also, active methods may rely on the deployment of an agent within the environment to get information about the individual nodes [9].

Passive methods detect symptoms from existing data sources, e.g., traffic metadata [12], traffic capture files, network log files [15], performance counters. Passive methods can utilize the data commonly provided by various network monitoring systems.

Some systems combine passive traffic monitoring to detect faults with active probing to determine the cause of failure. Identifying anomalies related to network faults and linking them with possible causes commonly utilizes some of the following approaches:

Inference-based approach uses a *model* to identify the dependence among components and to infer the faults using a collection of facts about the individual components [9], [17].

Rule-based approach uses *predefined rules* to diagnose faults [10]. The rules identify symptoms and determine how these contribute to the cause. The rules may be organized in a collaborative environment for sharing knowledge between administrators [7]. Kim et al. [18] propose a 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. De Paola et al. [19] deals with a 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. Dong and Dulay [20] developed an 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 [5].

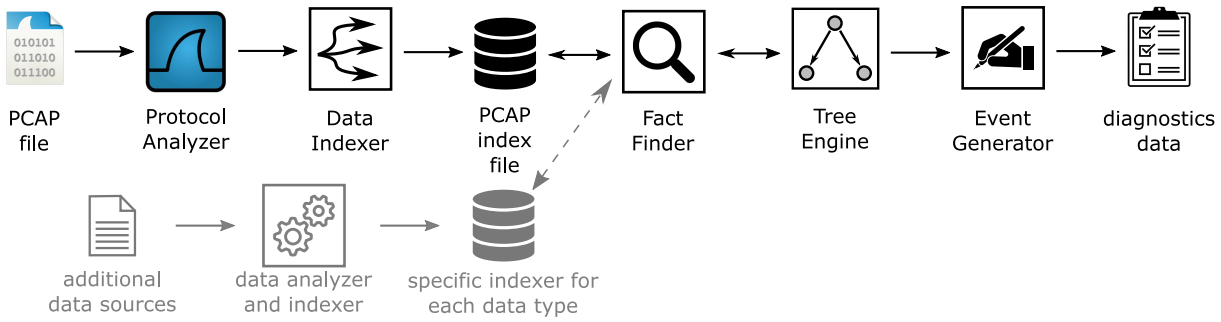

Figure 1. The top-level architecture of the proposed system. The architecture consists of five stages and one intermediate data storage (index file). The grey area represents optional architecture extensions — additional data sources.

Classifier-based approach *requires training data* to learn the normal and faulty states. The classifier can identify a fault and its likely cause [21]. Classifier-based methods were considered for misconfiguration detection in the home networks [22] and in the large network infrastructure [23]. *Tranalyzer* [24] is a flow-based analyzer that does traffic mining and a statistical analysis for large-scale networks. Big-DAMA [25] is a novel framework for detection and diagnosis of network traffic anomalies.

Network diagnostics based on traffic analysis can also use methods proposed for anomaly detection as some types of faults result in network communication anomalies.

Compared to other rule-based solutions, our system uses decision trees, which allows us to define more complex situations. Compared to simple rules (as used, for example, by a fishbone diagram), it is possible to make decisions based on previous diagnostic steps. Another difference is that our system does not need to know in advance what is wrong or what to focus on. Also, our system is not limited to only one type of data, and diagnostic rules are understandable by real administrators (not just scientists and programmers).

IV. Proposed System Architecture

We have built an expert system for analyzing network traffic, that has already been integrated into a worldwide business product [26]. The system combines rule-based and inference-based methods as it is easy to understand for network administrators. While the use of classifier-based methods has been proven very suitable for anomaly detection it lacks the capability to provide additional information for the detected case. The advantage of learning from provided data can only be exploited if a large set of annotated data is available. Contrary, the rule-based method can be extended also for detecting rare cases. The system only requires captured network traffic containing enough information about the event. Thus it is a completely passive method. Active methods generate additional traffic into a network (which can be unwanted in some situations) and require access to the network.

The proposed system is a processing pipeline that consists of several stages, as shown in Figure 1. The first stage, labeled as *Protocol Analyzer*, filters and decodes input packets using an external tool. The second stage takes decoded packets and converts them into a format for easier and faster data access (PCAP index file). The third stage, named *Fact Finder*, executes simple rules to identify facts significant from the diagnostics point of view. In the fourth *Tree Engine* stage, the decision tree utilizes the *Fact Finder* and identifies the possible problem cause. The fifth, and the last, stage *Event Generator* generates diagnostic outputs that contain detected errors and suggested solutions. Stages three, four, and five are easily extendable by the administrator who can add new rules and definitions.

The system can also be extended to use different data sources (e.g., log files or NetFlow records), as shown on the second row in Figure 1. Each data source requires specific data preprocessing that leads to the creation of an index file. The common part starts with the *Fact Finder* that can search indexed data of different data sources. If not specified otherwise, in the rest of the paper, we describe and evaluate the system only for a single data source represented by captured packet traces.

A. Protocol Analyzer

The first step in the processing pipeline is decoding captured network traffic in the PCAP format into a readable JSON format. We employ the tool TShark, which is a command-line version of the widely-used network protocol analyzer Wireshark. Because TShark follows the field naming convention used by Wireshark, we can use Wireshark Display Filter Expressions to select packet attributes. TShark supports all packet dissectors available in Wireshark. An example of TShark's output format with some omitted data is displayed in Figure 2.

```
{
...
  "_source": {
    "layers": {
      "frame": {
        "frame.number": "15",
        "frame.len": "84",
...
      "ip": {
        "ip.ttl": "50",
        "ip.proto": "6",
...
      "tcp": {
        "tcp.srcport": "25",
        "tcp.dstport": "1470",
...
      "smtp": {
        "smtp.response.code": "235",
...
}
```

Figure 2. An output from the TShark's JSON format.

Using TShark brings the following benefits:

- many protocol dissectors are available and the community quickly provides a parser for an emerged protocol;
- tunneled, segmented and reassembled data are support;
- data presentation is consistent with the Wireshark, which allows the creation of an easy-to-read API for diagnostics.

TShark provides not only data of fields from supported network protocols but also some computed data, such as round trip time, missing or retransmitted TCP segments, which can be used in diagnostic rules.

Even if our primary use case is to diagnose problems inside the captured network data, we would like to test that our system can work with other data sources as well. For this test, we have chosen to use the log files. Because each application has its own format of log messages and we were not able to find a universal tool that can parse the content of any log message into a JSON object, we have implemented a custom parser.

Our data preparation script takes log records one by one, and if a record matches some of the predefined regular expressions, the record is converted into a JSON format, as shown in Figure 3. Currently, only a few applications are supported - postfix, dovecot, and fail2ban. The output JSON format has the same structure as the JSON from the TShark tool, so future processing will remain the same.

```
Feb 20 01:12:19 mail dovecot: auth: passwd-file(info,
185.36.81.57): unknown user (SHA1 of given password:
ece4e6)
```

```
{
   "time": "1582161139",      # Feb 20 01:12:19
   "service": "mail dovecot",
   "mode": "auth",
   "username": "info",
   "ip": "185.36.81.57",
   "description": "unknown user"
}
```

Figure 3. Conversion of a single log record into a JSON object.

B. Data Indexer

Data Indexer converts data from the JSON format into a format suitable for fast searching by packets' field names (attributes) and their values. Most of the time, it will not be necessary to process the packets one by one, which significantly improves the resulting diagnostic speed. Each input packet is indexed and the following data is stored:

1) the packet itself;
2) a set of all field names of the packet;
3) map of values assigned to each packet's field.

Figure 4 shows an example of how indexing works. The entire index is represented by an associative array. First, the packet is stored under the *_raw* key and a packet number (in this case, 3). Using this value, it is possible to retrieve the packet in the same format as returned by TShark. Subsequently, the packet number is stored under a set of all indexed packets stored under the *_packets* key. This set will simplify some operations, and its usage can be seen in Figure 5. In the next steps, for each packet attribute and its values (each attribute can contain multiple values) a set of packet numbers is created (when it does not already exist). After that, the current packet number is added to the set.

```
1 index = dict() # associative array in
    Python
2 index["_raw/3"] = {"frame.number":
    ["3"], "dns.id": ["0x00007df5"],
    "ip.addr": ["192.168.1.1",
    "192.168.1.100"],...} # under the
    _raw + packet number key, the
    original packet in the JSON format
    is stored
3 index["_packets"] = {1, 2, 3} #
    _packets index contains set of all
    packet numbers
4 index["dns.id"] = {3} # all packets
    with any "dns.id" field value are
    saved under the field name key
5 index["dns.id/0x00007df5"] = {3} #
    packets which contain "dns.id" field
     with value "0x00007df5" are saved
    under field name/field value key
6 index["ip.addr"] = {1, 2, 3}
7 index["ip.addr/192.168.1.1"] = {1, 3}
8 index["ip.addr/192.168.1.100"] = {3}
```

Figure 4. An example of the indexing of a few fields from the DNS packet. The bold text is showing index keys and values, which have been added because of the new packet.

C. Fact Finder

The *Fact Finder* aims to identify specific situations useful for network diagnostics. Facts can be attributed to one or more packets, which are in some relation. For example, a successful DNS name resolution is a fact that consists of a query and a corresponding reply DNS messages. The facts are specified by rules describing which packets should be found and which relation they should fulfill. The format of these rules is described in Subsection V-B. A rule can consist of up to three parts:

1) a list of packet filters;
2) a list of assertions to express relation constraints;
3) parameters for the filters and assertions.

The system evaluates rules as follows: (i) Parameters are replaced by provided values. (ii) Each packet filter returns a list of packets matching the filter. (iii) Assertions are evaluated to select sets of packets satisfying the constraints. A result has the form of a collection of sets of packet numbers, e.g., a rule that identifies DNS request-reply pairs checks that the transaction ID in both the request and reply packets match. The last step is converting sets of packet numbers into lists of packets. Packets in lists are ordered by the packet numbers.

Filter expressions use Wireshark's display filter language. By using this language, the expression can be first tested in Wireshark before it is used in a *Fact Finder* rule. Assertion constraints use our created language that is based on the Wireshark's display filter language. There are three changes made to the original language, which add support of:

1) working with packets from filter expressions;
2) simple math operations (+, -, *, /);
3) parameters for expressions. The parameters do not increase language capability but aim to simplify rule definition.

The evaluation of the facts begins with searching for packets with specific attributes. However, this varies depending on how these attributes are specified. In the case of a simple condition, it is possible to use the index created in the *Data Indexer* step, but with more complex conditions, this is not possible. A more complex condition is one that contains either a regular expression, function (string length, substring), or some comparison ($<, <=, >, >=$).

If it is possible to search for packets using the created index, the appropriate packet numbers are searched for using each attribute specified. Based on the specific relation between attributes, the adequate set operation is applied to the sets of packets. This process is shown in Figure 5. This figure describes finding packets by using the created index.

```
1  search: dns
2  result = index["dns"]
3
4  search: dns.flags.response == 1
5  result = index["dns.flags.response/1"]
6
7  search: dns and ip.addr == "10.10.10.1"
8  dns_packets = index["dns"]
9  ip_packets=index["ip.addr/10.10.10.1"]
10 result = dns_packets.intersection(
       ip_packets) # packets both in
       dns_packets and ip_packets
11
12 search:smtp.response.code != 250
13 all_packets = index["_packets"]
14 skip_packets = index["smtp.response.
       code/250"]
15 result = all_packets.difference(
       skip_packets) # packets in
       all_packets but not in skip_packets
```

Figure 5. An example of index usage when searching for packets that meet the specified constraints. When combining attributes in constraints, results from individual attributes are combined using set operations. The bold text shows the packet specification.

In case the packet specification cannot be evaluated using the created index, it is necessary to go through each packet and evaluate the condition for its values. This is accomplished by replacing the attributes in the expression (e.g., *smtp.response.code matches "[45] [0-9] [0-9]" and ip.addr = "10.10.1.1"*) with values from each packet. Because a list of values represents each attribute's value, the evaluation process must try all value combinations. If at least one combination fulfills the packet specification, the packet is added to the set of fulfilling packets. The principle is shown in Figure 6. Because there was not such a complicated rule in DNS protocol, we are showing this principle on the SMTP rule.

After all packets have been searched, they are represented only by a list of packet numbers. Before working with the packet's data, it is necessary to replace these packet numbers with the actual packets. After that, constraints defining packet relationships can be evaluated (assert rules). An example of a relationship is a request-reply pair of packets that are linked together by a request ID. Searching for such packets is accomplished by creating all possible packet combinations (Cartesian product) and evaluating all conditions for each combination.

```
1  search packets: smtp.response.code
       matches "[45] [0-9] [0-9]" and
       ip.addr = "10.10.1.1"
2  import re # regular expression module
3  result = set()
4  def check_packet(packet):
5    values = {}
6    for value in packet["smtp.response.
         code"]: # only packets with field
         smtp.response.code are used
7      values["smtp.response.code"]=value
8      for value in packet["ip.addr"]: #
           only packets with attribute ip.
           addr are used
9        values["ip.addr"] = value
10       if re.search(values["smtp.
             response.code"],"[45][0-9]
             [0-9]") and values["ip.addr"]
             == "10.10.1.1":
11         result.add(packet_number)
12         return result
13
14 for packet_number in index["_packets"]:
15   packet = index["_raw/"+packet_number]
16   result = check_packet(packet, result)
17 return result
```

Figure 6. An example of finding all packets that meet the specified condition, which can not be evaluated by using the created indexes. When evaluating a condition, all value combinations are tested for each packet. The bold text shows the packet specification and the corresponding condition.

The principle of evaluating the assert conditions is shown in Figure 7. The code in the figure contains a $relation()$ function that combines all possible values (similar to the code in Figure 6) and compares whether at least one value combination meets the defined relation function (e.g., " $==$ " for equality). The $relation()$ function works with a *packets* dictionary that contains a list of packets that are saved under the keys defined in the facts section of the rule.

```
1  facts:
2    dns_query: dns.flags.response == 0
3    dns_reply: dns.flags.response == 1
4  asserts:
5   - dns_query[udp.stream] ==
        dns_reply[udp.stream]
6   - dns_query[dns.id] ==
        dns_reply[dns.id]
7
8  result = []
9  for query in packets["dns_query"]:
10   for reply in packets["dns_reply"]:
11     if relation(query["udp.stream"],
           "==", reply["udp.stream"])
12 and relation(query["dns.id"],"==",
       reply["dns.id"]): # relation()
       function checks all combinations of
       values from two lists
13       result.append({"dns_query": query
             , "dns_reply": reply})
14 return result
```

Figure 7. Example of a packet set search (DNS query and response) that meets the defined constraint (packets from the same UDP stream and the same request ID). The bold text is sharing assert constraints and the corresponding $relation()$ function.

D. Tree Engine

The tree engine infers the possible error cause by evaluating a decision tree that contains expert knowledge about supported network protocols and services. Each node of the tree contains a diagnostic question. Questions refer to facts identified by the *Fact Finder*. Based on the question's result, the next tree node is chosen. This node transition creates a path that begins in a root node and finishes in a leaf node. Paths in the tree represent gathered knowledge and lead to the possible cause of the problem.

The decision tree consists of declarative specifications of tree nodes enriched by Python code. The declarative part is responsible for creating the tree and consists of a rule name, a rule type, a *Fact Finder* rule, and two branches, which cover the success and the fail result of the *Fact Finder*. Both branches can define the next rule, which should be processed.

Python codes are located inside the success and fail branches. These codes are responsible for processing logic (e.g., saving packets for future tree nodes or translating error codes from packets into human-readable format) and generating diagnostic results. The format of rules is described in Subsection V-A.

E. Event Generator

During the diagnostic process, a report is created to provide diagnostic information for network administrators. The diagnostic report is produced in a human-readable format, as well as in a machine format useful for further processing or visualization. The report consists of events that are constructed in tree nodes based on the derived knowledge and processed packets. Each event describes one situation that happened in the network. For example, the connection to the HTTP server has been detected.

Each rule consists of a name, description, and severity of the detected situation. Additionally, the event may include a suggestion message and data from the provided packet. The provided packet is specified as a parameter in the tree rule. By using this packet, parameters such as flow identification or timestamp can be associated with the event. Subsection V-C describes the format of the event rules.

V. Rule Specification

The diagnostic engine defines each protocol as a decision tree. The tree consists of nodes representing administrator questions, and edges representing answers to these questions. The edge can move the diagnostic process from one question to another (within the same protocol or another) or finish the process with the discovered result.

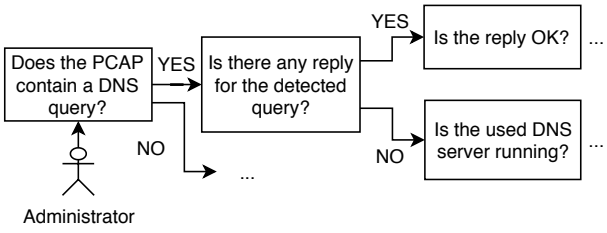

Figure 8. A simple illustration of a binary decision tree. An administrator diagnoses a DNS problem by anwering questions in the predefined order.

The questions simulate thinking of a real administrator. Typically, an administrator starts to search for certain network packet values and after the search for them is finished, the administrator searches for next values based on the result. In our solution, each question can only have two answers: success or failure. This yields a binary decision tree. Figure 8 shows an example of a small portion of the DNS tree.

The decision tree needs to be converted to a format understandable by our system. This conversion is split into three steps: 1) defining tree nodes (*Tree node* rules), 2) defining conditions for choosing tree nodes (*Fact Finder* rules) and 3) defining the diagnostic report (*Event definition* rules). The following subsections describe the syntax for each of these rules. The reason why a node rule does not contain a lookup code and an event definition directly and they need to be defined in separate rules is that multiple rules would not be able to use the same lookup code and events (increases reusability).

The conversion of the decision tree assigns a name to each tree node. We use the node names as labels for switching from one node to another. Each node tries to find specific facts, defined as a *Fact Finder rule*. Based on the condition, if some fact was found or not, the next diagnostic step is chosen. Each rule can have one or none success and fail branches. Branches contain executable Python code and the next node rule name. After the execution of the Python code, the analysis switches to the next node. Figure 9 shows the pseudocode for writing tree nodes.

```
1 tree_node_id:
2   if fact_finder_rule finds some facts:
3     success branch_code
4     jump to the next tree_node_id
5   else:
6     fail branch_code
7     jump to the next tree_node_id
```

Figure 9. Pseudocode for writing a tree node. Each node should have a unique id, lookup condition, and branch codes.

A. Tree Node Rules

All the rules are saved in a declarative YAML format. This format is easily understandable by programming code and by people without programming skills (we assume that not all network administrators are also programmers). Even if the system already contains some protocols, the administrators can easily add new protocols or can extend capabilities of the current protocols by updating the rules. In the following paragraphs, the format of the rules will be described. Names of the sections as they used in rules are placed inside the text.

The rule definition begins with the rule name (rule section *id*) and the execution of a *Fact Finder* rule (rule section *query*). The result of the *Fact Finder* is a list of associative arrays. Each array can contain multiple packets, where the packet name is the key to the array. These packets will be processed according to the *Tree Node* rule type (rule section *type*). The default behavior selects the first array from the list of arrays (the list is ordered by the arrival time of packets) and marks it as a found fact. The second type of rule is a "foreach" type, which iterates through the list of arrays and progressively marks each array as a fact and executes the defined rule. For example, the foreach rule can analyze each query to the server or each response to the selected query (as shown in Figure 10).

```
1  id: DNS query detected # name of the
       rule
2  query: exists DNS reply for the detected
       query? # Facts Finder rule
3  type: foreach
4  success:
5   state: is reply ok? # next state
6   code: | # Python code follows
7    reply_pkt = _fact["dns_reply"]
8    save("dns_reply", reply_pkt)
9    event("reply_detected", reply_pkt)
10 fail:
11  state: find any reply from the same
       destination server # next state
12  code: | # Python code follows
13   query_pkt = load("dns_query")
14   event("reply_not_detected",query_pkt)
```

Figure 10. Simple Tree Engine rule showing what should be done if a DNS query was detected.

Furthermore, the rule consists of two parts, with only one executed (depending on whether the diagnostic engine has found the searched fact or not). The format of both parts is the same. Each part consists of the name of the next rule with which the diagnostics should continue (rule section *state*) and the Python code (rule section *code*). Each rule can switch to to a rule from another protocol to diagnose problems across several protocols, e.g., if an SMTP communication is not detected, we will check if there are any ICMP unreachable messages, failed TCP connection attempts or incorrect DNS resolutions. If the next rule is not specified, the diagnostic engine stops the diagnostic process.

The Python code can process packet data, make logical decisions and most importantly, generate diagnostic output. Within the Python code, it is possible to use any Python 3 code and it is also possible to utilize the following variables and functions defined by the engine:

1) *fact* - contains the first fact found (or the next one in the foreach type rule)
2) *facts* - contains all the facts found
3) *save()* - saving any value for further processing (inside another Tree node rule or as a parameter in the *Fact Finder* rule);
4) *load()* - read the value previously saved by the save() function;
5) *event()* - generates a diagnostic report, where the parameter is a packet to which the report refers.

Figure 10 shows an example of a rule defining the middle node from the tree in Figure 8. The figure shows a node describing that a DNS query has been detected (*id*) and the rule is looking for a DNS response for the detected query (*query*). For each detected reply, a successful section (*success*) is executed (foreach *type*). The response is saved, the diagnostic message is generated (*code*), and the diagnostic process continues to check whether the response is without error (*state*). If no response to the query is found, the failure section is executed (*fail*). First, the original query is retrieved, the diagnostic message is generated (*code*), and then the diagnostic process continues with the next state (*state*).

B. Fact Finder Rules

Rules in this section describe how a question is converted into packet lookup functions. Each rule may look for several independent packets, which are combined and checked if their relation fulfills assert conditions. Each question returns a list of associative arrays, where each arrays represents a unique combination of packets fulfilling the assert conditions (packet names are arrays keys).

Each rule needs to have a name (rule section *id*), which is used inside the *Tree Node* rules. The rule can have parameters to make rules more reusable (rule section *params*). For example, instead of creating a rule for each error code, it is possible to create one rule with a parameter containing the expected error code. Parameters are used as variables in the following sections and can have a format of a single value or single packet (previously saved by *save()* function within the *Tree Node* rule). Rule section *facts* define the name of the searched packets and the filter. The filter uses Wireshark display language and specifies which packets should be assigned to the specified name (each packet can be assigned to multiple packet names). Rule section *asserts* define conditions for the detection of packets. The conditions use our custom language, which is based on the Wireshark display language. However, the custom language allows to:

- use math operations addition(+), subtraction(-), multiplication(*) and division(/);
- use single value parameters as if the values were directly inserted into the condition;
- use values from packets (provided from packets or params section). The format is packet_name[field.name], where field.name is the Wireshark name assigned to the attribute.

Figure 11 shows an example of a simple rule for the question *Is there any DNS reply for the detected DNS query?* After the rule name (*id*), the parameter *dns_query* for the assert conditions is specified (*params*). The rule contains a definition of the *dns_reply* packet (*packets*), which is used in the assert conditions. The conditions (*asserts*) are checking whether the *dns_reply* belongs to the same UDP stream as the provided *dns_query* and if the reply packet is answering to the specific query.

```
1  id: exists DNS reply for the detected
       query?  # name of the rule
2  params: # saved data from any tree rule
3   -dns_query
4  facts: # which packets we are looking
       for
5   -dns_reply: dns.flags.response == 1
6  asserts: # packets relation constrain
7   -dns_query[udp.stream]==dns_reply[udp.
       stream]
8   -dns_query[dns.id]==dns_reply[dns.id]
```

Figure 11. Example of a DNS fact rule for checking if the PCAP file contains a reply for the provided query or not.

By default, the *Fact Finder* rules are working with the data saved inside the PCAP index file. To allow searching for facts from different data index files, it is necessary to specify the data type (rule section *type*). Figure 12 shows an example of such a rule that looks for a log record containing specific values related to the wrong username event. Because the rules are

using the same *Tree Node* engine, the rule of one type can use parameters from a different type of rule. In our example, we are looking for a record with the same IP address as used in the provided *imap_query* packet.

```
id: wrong username?
params: # saved packet from previous
    rules
 -imap_query
type: log # the data are saved in
    different index file
facts:
 -auth: description == "unknown user"
     and ip == imap_query[src.ip]
```

Figure 12. Fact rule for different data source (log file). The rule is checking presence of a specific record in input log file.

C. Event Definitions

Event rules describe how the diagnostic message will look. The message is created by calling a function *event()* from the *Tree Node* rule. In addition to the event name, the event function also accepts a packet parameter (the event is related to this packet). The idea is that from the provided packet, the time, the flow identification, and possibly other specified values are extracted and inserted into the message.

Each rule consists of a name, severity, description, instructions on how to fix the problem, and a list of fields from the provided packet. The field list contains the names according to the Wireshark terminology. An optional part of each field is its description to help the administrator understand its value. The description and suggestion may contain variables. The variables are written as $\{fieldname\}$, and will be replaced by values from the provided packet when the diagnostic report is generated.

Figure 13 shows an event describing the error that no DNS response was detected for the DNS query. From the provided packet, the queried domain name is inserted into the description and the DNS server address into the suggestion. Additional items will be also included in the output: transaction ID, queried domain name, and DNS server IP address.

```
id: reply_not_detected
severity: error
description: "No reply for query '{dns.
    qry.name}' has been detected."
suggestion: "Check if the DNS service is
     running on {ip.dst}. If yes, check
    the firewall on the server and the
    path between server and the client."
fields:
- name: dns.id
  description: Transaction ID
- name: dns.qry.name
  description: Queried domain name
- name: ip.dst
  description: Server IP address
```

Figure 13. A DNS event example, that reports that the query wasn't detected.

VI. Use Case and Evalution

The goal of the tool is on-demand diagnostic of the selected network traffic. It is essential to note that the goal is not an on-line (24/7) analysis or analysis of a large amount of data. The idea is based on a use case that when an administrator detects a problem on the network, it triggers a capture on the selected network traffic. For example, if a client with the address 192.168.0.20 is unable to establish a TLS connection with the server on the address 192.168.0.1, the administrator will capture the communication between these two stations.

We have implemented diagnostic rules for several application and service protocols. Table I shows the current list of supported protocols and their complexity in term of *Tree node* and *Fact Finder* rule count, and their capabilities in term of *Event* rule count.

Table I. Supported protocols and amount of rules and success, warning, error events which describe various protocol behavior situations.

Protocol	Tree nodes	Fact finders	Events		
			Success	Warning	Error
DHCP	24	22	10	9	4
DNS	12	12	8	4	5
FTP	24	10	15	6	7
HTTP	3	3	2	1	1
ICMP	4	2	0	0	4
IMAP	15	8	7	3	9
POP	21	7	5	10	7
SIP	38	22	15	1	8
SLAAC	8	7	1	6	1
SMB	27	25	20	3	5
SMTP	17	13	9	6	9
SSL	2	2	2	0	1
TCP	10	10	0	7	2

We have tested the functionality and performance of the implemented tool. Table II provides a sample of data from performance experiments. The execution time is divided into TShark tool processing time, time used for indexing the JSON from TShark, and the analysis time. Five files of different sizes containing some representative data are presented. The tests were performed on a CPU Intel Xeon Silver 4116 2.10 GHz and 4 GB RAM. It should be noted that only one CPU core was used for the diagnostics on the given processor, as TShark as well as the implemented tool are single-core applications.

Table II. The table shows the diagnostics execution time for the selected PCAP files of different sizes.

Size [MB]	Packets	Flows	Time [s]			
			TShark	Indexes	Analysis	Total
182.253	222 372	7155	18.717	27.407	19.678	65.802
21.569	62 471	7002	5.004	5.940	7.748	18.692
9.640	14 509	954	1.969	1.533	1.464	4.966
1.687	3 544	373	1.295	0.589	1.186	3.070
0.978	4 848	84	0.821	0.669	1.291	2.781

The output of the tool is a report in JSON format, which enables easy machine processing. We have created a web interface to visualize the report in a more human-readable format. The visualization consists of two parts. The first part shows a list of all detected events. After clicking on any event, the detail of this event is displayed in the second part. Below an event name, a suggestion for fixing the problem is displayed in which real values from the packet have replaced the message variables (written in curly brackets). After the suggestion message, the rest of the event attributes are displayed.

To demonstrate the functionality of the tool, we have diagnosed a PCAP file that was captured on a station with the IP address 10.10.1.4. The tool diagnoses all predefined protocols and displays the detected events in a hierarchical structure. Figure 14 shows this situation, with some events omitted for simplicity. For each event, a description and an icon of the event are displayed. The highest severity is then propagated from the deepest events out so that it is possible to find problem situations in a large number of events quickly. In addition to the detected error with the DNS response, it is possible to see other communications in the picture, which were without error.

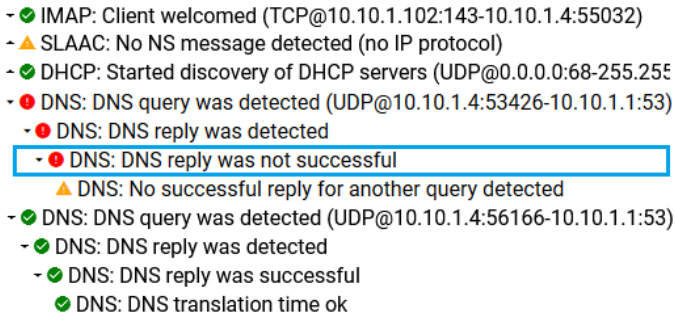

Figure 14. The figure shows a list of detected events for a diagnosed PCAP file. The list contains events from multiple protocols with different severities. The screenshot was taken from the *Flowmon Packet Investigator*, a product that has integrated the proposed tool.

After selecting an event from the list (represented by a blue rectangle), a detailed description of the event is displayed. Figure 15 shows this output, which describes the reason why the domain name translation failed. In addition to the event name, the listing is divided into three sections: 1) suggestions for the administrator on how to fix the error, 2) a brief summary of the event (description, severity, and flow), and 3) attributes extracted from the packet that triggered the displayed event.

DNS reply was not successful

Check if the domain 'naps.google.com' is correctly specified (e.g., some typo error) and check the DNS server '10.10.1.1' configuration.

Description	Failure reply with code '3' has been detected.
Protocol	DNS
Severity	error
Flow	UDP@10.10.1.1:53-10.10.1.4:53426
decoded error code	Domain name does not exist
Frame time epoch	12.11.2016 17:32:23
Frame number	2
IP version	4
IP source	10.10.1.1
IP destination	10.10.1.4
IP proto	17
UDP source port	53
UDP destination port	53426
dns.id	0x00000c01
dns.qry.name	naps.google.com
dns.flags.rcode	3

Figure 15. The figure shows an example of diagnostic output for a DNS error. It suggests to an administrator to check the domain name and the server configuration. The screenshot was taken from the *Flowmon Packet Investigator*, a product that has integrated the proposed tool.

VII. Conclusion

Network troubleshooting can be a nightmare for administrators because of system complexity. There may not be an evident link between the issue reported by a user and the real cause of the problem. Some of the errors can be identified and analyzed by examining network traffic. However, using the traditional mostly manual approach is time-consuming and requires significant expertise. The presented paper describes the automatized approach to network traffic analysis able to identify errors using the rule-based approach. The rules encode expert knowledge and are evaluated for the captured traffic.

While the rule-based approach may be considered as an old-fashioned approach these days when the majority of research considers a machine learning-based approach, it was demonstrated that knowledge encoded in the form of rules provides an efficient method for network troubleshooting. Moreover, because of expert-designed rules, it is possible to add information that explains the possible cause of the issue and recovery options. Mainly the explainability associated with this approach seems to be the biggest benefit for users. The drawback, of course, is related to the necessity of creating and testing the rules base. Also, the method is susceptible to the quality of data sources. When the captured communication is incomplete, the method can provide incorrect results.

The performance is important for any method to be practically usable. The presented method uses a rule evaluation engine that traverses decision trees for problem domains, which can be evaluated in a reasonable timeframe. However, as nodes of the tree contain expressions that can potentially require complex operations over the input data, the set of indexes is precomputed to improve the performance.

The proof-of-concept demonstrating the approach was implemented and further finalized to the tool integrated into the commercial suite for network monitoring. The tool is commercially provided on the market by Flowmon Networks company as Flowmon Packet Investigator [26].

Future work will focus on:

- adding support of new protocols, e.g., NTP or SNMP;
- even though the current performance is good enough, it can always be better, and our goal will be to decrease the execution time of the diagnostic process;
- because the quality of the diagnostic output highly depends on the quality of the input data, we would like to create a validation technique (maybe by using machine learning techniques) to check the validity of the input data (e.g., detection of packets loss);
- after separating the processing of the input data from the Fact Finder into Data Indexer, it is now possible to create a distributive solution that consists of many data collection points across the network. At each point, the data would be indexed by the Data Indexer and sent to the central processing unit.

Acknowledgment

This work was supported by project "Network Diagnostics from Intercepted Communication" (2017-2019), no. TH02010186, funded by the Technology Agency of the Czech Republic, the BUT FIT grant FIT-S-20-6293, "Application of AI methods to cyber se-curity and control systems", and by private network monitoring company Flowmon Networks.

REFERENCES

[1] M. Holkovič and O. Ryšavý, "Network diagnostics using passive network monitoring and packet analysis," The Fifteenth International Conference on Networking and Services (ICNS), 2019, pp. 47–51.

[2] R. Wang, D. Wu, Y. Li, X. Yu, Z. Hui, and K. Long, "Knight's tour-based fast fault localization mechanism in mesh optical communication networks," Photonic Network Communications, vol. 23, no. 2, 2012, pp. 123–129.

[3] M. Solé, V. Muntés-Mulero, A. I. Rana, and G. Estrada, "Survey on models and techniques for root-cause analysis," arXiv preprint arXiv:1701.08546, 2017.

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

[5] M. łgorzata Steinder and A. S. Sethi, "A survey of fault localization techniques in computer networks," Science of computer programming, vol. 53, no. 2, 2004, pp. 165–194.

[6] C. Guo et al., "Pingmesh: A large-scale system for data center network latency measurement and analysis," in ACM SIGCOMM Computer Communication Review, vol. 45, no. 4. ACM, 2015, pp. 139–152.

[7] B. Agarwal, R. Bhagwan, T. Das, S. Eswaran, V. N. Padmanabhan, and G. M. Voelker, "Netprints: Diagnosing home network misconfigurations using shared knowledge," in Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation, ser. NSDI'09. Berkeley, CA, USA: USENIX Association, 2009, pp. 349–364.

[8] L. Lu, Z. Xu, W. Wang, and Y. Sun, "A new fault detection method for computer networks," Reliability Engineering & System Safety, vol. 114, 2013, pp. 45–51.

[9] S. Kandula et al., "Kandula, srikanth and mahajan, ratul and verkaik, patrick and agarwal, sharad and padhye, jitendra and bahl, paramvir," ACM SIGCOMM Computer Communication Review, vol. 39, no. 4, 2009, pp. 243–254.

[10] M. Luo, D. Zhang, G. Phua, L. Chen, and D. Wang, "An interactive rule based event management system for effective equipment troubleshooting," in IECON 2011-37th Annual Conference on IEEE Industrial Electronics Society. IEEE, 2011, pp. 2329–2334.

[11] A. Mohamed, "Fault detection and identification in computer networks: A soft computing approach," Ph.D. dissertation, University of Waterloo, 2010.

[12] D. Brauckhoff, X. Dimitropoulos, A. Wagner, and K. Salamatian, "Anomaly extraction in backbone networks using association rules," in Proceedings of the 9th ACM SIGCOMM conference on Internet measurement. ACM, 2009, pp. 28–34.

[13] L. Benetazzo, C. Narduzzi, P. A. Pegoraro, and R. Tittoto, "Passive measurement tool for monitoring mobile packet network performances," IEEE transactions on instrumentation and measurement, vol. 55, no. 2, 2006, pp. 449–455.

[14] K.-H. Kim, H. Nam, J.-H. Park, and H. Schulzrinne, "Mot: a collaborative network troubleshooting platform for the internet of things," in Wireless Communications and Networking Conference (WCNC), 2014 IEEE. IEEE, 2014, pp. 3438–3443.

[15] T. Qiu, Z. Ge, D. Pei, J. Wang, and J. Xu, "What happened in my network: mining network events from router syslogs," in Proceedings of the 10th ACM SIGCOMM conference on Internet measurement. ACM, 2010, pp. 472–484.

[16] M. Vásquez-Bermúdez, J. Hidalgo, M. del Pilar Avilés-Vera, J. Sánchez-Cercado, and C. R. Antón-Cedeño, "Analysis of a network fault detection system to support decision making," in International Conference on Technologies and Innovation. Springer, 2017, pp. 72–83.

[17] S. Jamali and M. S. Garshasbi, "Fault localization algorithm in computer networks by employing a genetic algorithm," Journal of Experimental & Theoretical Artificial Intelligence, vol. 29, no. 1, 2017, pp. 157–174.

[18] S. Kim, S. j. Ahn, J. Chung, I. Hwang, S. Kim, M. No, and S. Sin, "A rule based approach to network fault and security diagnosis with agent collaboration," in Artificial Intelligence and Simulation, T. G. Kim, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 597–606.

[19] A. De Paola, S. Fiduccia, S. Gaglio, L. Gatani, G. Lo Re, A. Pizzitola, M. Ortolani, P. Storniolo, and A. Urso, "Rule based reasoning for network management," in Seventh International Workshop on Computer Architecture for Machine Perception (CAMP'05), July 2005, pp. 25–30.

[20] C. Dong and N. Dulay, "Argumentation-based fault diagnosis for home networks," in Proceedings of the 2nd ACM SIGCOMM Workshop on Home Networks, ser. HomeNets '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 37–42. [Online]. Available: https://doi.org/10.1145/2018567.2018576

[21] E. S. Ali and M. Darwish, "Diagnosing network faults using bayesian and case-based reasoning techniques," in Computer Engineering & Systems, 2007. ICCES'07. International Conference on. IEEE, 2007, pp. 145–150.

[22] B. Aggarwal, R. Bhagwan, T. Das, S. Eswaran, V. N. Padmanabhan, and G. M. Voelker, "NetPrints: Diagnosing home network misconfigurations using shared knowledge," Proceedings of the 6th USENIX symposium on Networked systems design and implementation, vol. Di, no. July, 2009, pp. 349–364. [Online]. Available: http://portal.acm.org/citation.cfm?id=1559001

[23] M. Chen, A. Zheng, J. Lloyd, M. Jordan, and E. Brewer, "Failure diagnosis using decision trees," International Conference on Autonomic Computing, 2004. Proceedings., 2004, pp. 36–43. [Online]. Available: http://ieeexplore.ieee.org/document/1301345/

[24] S. Burschka and B. Dupasquier, "Tranalyzer: Versatile high performance network traffic analyser," in 2016 IEEE Symposium Series on Computational Intelligence, SSCI 2016, 2017.

[25] P. Casas, T. Zseby, and M. Mellia, "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.

[26] "Flowmon products overview," https://www.flowmon.com/en/overview, accessed: 2020-May-27.