Message Format and Field Semantics Inference for Binary Protocols Using Recorded Network Traffic

Gergő Ládi, Levente Buttyán, Tamás Holczer
Laboratory of Cryptography and System Security
Department of Networked Systems and Services
Budapest University of Technology and Economics
Budapest, Hungary
{gergo.ladi, buttyan, holczer}@crysys.hu

***Abstract*—Protocol specifications describe the interaction between different entities by defining message formats and message processing rules. Having access to such protocol specifications is highly desirable for many tasks, including the analysis of botnets, building honeypots, defining network intrusion detection rules, and fuzz testing protocol implementations. Unfortunately, many protocols of interest are proprietary, and their specifications are not publicly available. Protocol reverse engineering is an approach to reconstruct the specifications of such closed protocols. Protocol reverse engineering can be tedious work if done manually, so prior research focused on automating the reverse engineering process as much as possible. Some approaches rely on access to the protocol implementation, but in many cases, the protocol implementation itself is not available or its license does not permit its use for reverse engineering purposes. Hence, in this paper, we focus on reverse engineering protocol specifications based solely on recorded network traffic. More specifically, we propose a method that can infer protocol message formats as well as certain field semantics for binary protocols from network traces. We demonstrate the usability of our approach by running it on packet captures of two known protocols, Modbus and MQTT, then comparing the inferred specifications to the known specifications of these protocols.**

***Index Terms*—protocol reverse engineering, message format, field semantics, inference, binary protocols, network traffic, Modbus, MQTT**

I. Introduction

Protocols describe the formats, types, contents, and sequence of messages that are sent and received in order to exchange data between the communicating parties, as well as the rules according to which these messages must be processed. The protocols themselves are defined in specifications, which are not always available to the general public. This is unfortunate, as having access to specifications is required for the generation of models that serve as the basis of several security-related applications, such as the development of intrusion detection systems (IDS) that understand the protocol and can raise alarms when anomalous protocol messages are detected [1], the creation of protocol-specific honeypots that simulate a device running said protocol for attacker behaviour analysis [2], and fuzz testing protocol implementations for programming errors or hidden features [3].

Protocol reverse engineering is an area of study that provides methods which aim to reconstruct the specifications for protocols where these are not available. Given that manual reverse engineering of protocols is rather time consuming, and that new protocols appear frequently, it is generally recommended that an automated approach be used. These aim to provide at least partial information about protocols in at least a semi-automated fashion, typically relying on the analysis of captured network packets or existing protocol implementations (binaries), or a combination of these [4]. However, protocol implementations may not always be available, and licensing restrictions or user agreements may forbid such reverse engineering. For this reason, we focus on methods that only rely on captured network traffic.

The reverse engineering process is usually comprised of three main phases [5]. The first phase involves setting up the environment in which the analysis will be conducted, as well as performing the necessary preparation steps such as generating and capturing network traffic, and instrumenting binaries. The second phase focuses on determining the types of the possible messages (i.e. messages that result in functionally distinct behaviour from the other party) along with the semantics of the fields (groups of bytes) within the messages. The third phase focuses on constructing a state machine for the protocol, which describes the valid sequences of the previously determined message types (i.e. the grammar of the protocol). To measure the goodness of the inferred specifications, typically three metrics are used: correctness, conciseness, and coverage [4], where correctness measures what percentage of the inferred messages represent true messages, conciseness shows how many inferred messages represent one true message, and coverage shows what portion of the true messages were found.

Protocols can be classified into two groups: plain text and binary. Plain text protocols such as Hypertext Transfer Protocol (HTTP) or Simple Mail Transfer Protocol (SMTP) exchange human-readable messages where the fields are separated by delimiters such as spaces, colons, or new line characters, and at least one field contains a keyword that determines how the message should be interpreted. On the

The research presented in this paper has been partially supported by the Hungarian National Research, Development and Innovation Fund (NKFIH, project no. 2017-1.3.1-VKE-2017-00029), and by the IAEA (CRP-J02008, contract no. 20629). The first author has also been supported by the European Union, co-financed by the European Social Fund (EFOP-3.6.2-16-2017-00013).

other hand, binary protocols such as Server Message Block (SMB) or Modbus exchange binary messages that are not human-readable, lack field separators, and one or more groups of bytes determine how the message should be interpreted.

In this paper, we present a novel graph-based algorithm which can infer not only the message types, but also some field semantics of binary protocols, using nothing but the statistical properties of recorded network traffic. We have implemented and tested the algorithm on real-world captures of two commonly used binary protocols, Modbus and MQTT, achieving perfect correctness and completeness scores as well as decent conciseness scores that surpass those of existing state-of-the-art methods. We do not currently aim to reconstruct the state machine of the protocol.

The rest of the paper is structured as follows: in Section II, we discuss related work. In Section III, we present our algorithm in detail, along with additional possible optimization steps. Next, in Section IV, we evaluate the previously presented algorithm on packet captures of two common protocols, Modbus and MQTT. Then, in Section V, we briefly discuss the possible limitations of our solution, followed by opportunities for future work. Finally, Section VI concludes our paper.

II. Related Work

Protocol reverse engineering dates back to the 1950s, where it typically meant the analysis of finite state machines for fault detection [6]. The first well-known project that aimed at restoring the specifications of a computer protocol was the Protocol Informatics Project by M. A. Beddoe [7] in 2004, which used bioinformatical algorithms on network traces to infer the message types of the text-based protocol HTTP. It was later followed by Discoverer, Biprominer, ReverX, ProDecoder, and AutoReEngine [8]–[12] that all relied only on network traffic. Some explicitly targeted text-based protocols, some targeted binary protocols, while others considered both. Methods relying on reversing implementations appeared under the names of Polyglot, AutoFormat, and ReFormat [13]–[15]. Solutions to reverse the protocol grammar have also been proposed in the form of ScriptGen, Prospex, Veritas, and MACE [16]–[19].

In this paper, we aim to compete with Discoverer, Biprominer, and ProDecoder, three different approaches for reversing binary protocols. Their details, as given by their authors, can be seen in Table I.

TABLE I
Performance metrics of the three approaches

Approach	Correctness	Conciseness	Coverage	Tested on # protocols
Discoverer	0.9	5	0.95	3
Biprominer	0.99	Unknown	0.967	3
ProDecoder	0.975	Unknown	0.975	2

Discoverer is based on tokenization, recursive clustering, and merging. Biprominer relies on learning, labeling, and building a transition probability model, while ProDecoder uses n-gram generation, keyword identification, clustering, and sequence alignment. Our algorithm is based on a sequence of graph construction and optimizations.

III. Our Approach

Our approach consists of four distinguishable phases. The first phase is a preparation phase, in which data is gathered and transformed such that it can be processed in the second phase. The second phase is the core algorithm that constructs an acyclic connected graph (tree) based on the input. In the third phase, optional optimizations may be run on the tree. These optimizations generally improve a certain metric at a possible cost of a different metric. Finally, the tree is used to enumerate the inferred message types and field semantics.

A. Preparations

In the preparation phase, the environment needs to be planned and set up. In order to observe and record protocol traffic, at least one client and at least one server application instance (or in the case of peer-to-peer applications, two instances) should be running. These instances may or may not be running on the same device, and if multiple devices are used, these need not be of the same type (e.g. one can be an ordinary computer, while the other an industrial programmable logic controller (PLC)). This approach needs no access to the source code or the compiled application binaries, nor does it need access to the memory of the devices where these are running. The only requirement is that there has to be a way to monitor and capture network traffic flowing between the application instances. This is typically done by running *tcpdump* or *Wireshark* on one of the devices.

Once the environment is set up and the capture is running, traffic should be generated by invoking as many features of the client with as many different options and in as many different combinations as possible, all repeated a number of times. This ensures that most of the message space is covered, which is essential for near-complete and accurate recovery of the protocol specification.

B. Constructing a Tree

Each captured packet is read into the memory. For each packet, a pointer is assigned that initially points to the first byte of the packet. This pointer is later used to keep track of how many bytes have already been processed in that specific packet. A separate pointer is needed for each packet as some steps of the algorithm increment this pointer by different amounts for different packets.

The algorithm maintains and builds a graph that initially consists of one node, the root node (which also is a leaf at this point). In each step, new nodes of different colours are appended to one of previous leaves. The colours are used to indicate the inferred field semantics, and are based on the following decisions:

1) Constants - Check the next byte of each packet. If this is the same for all packets, consider this byte a constant.

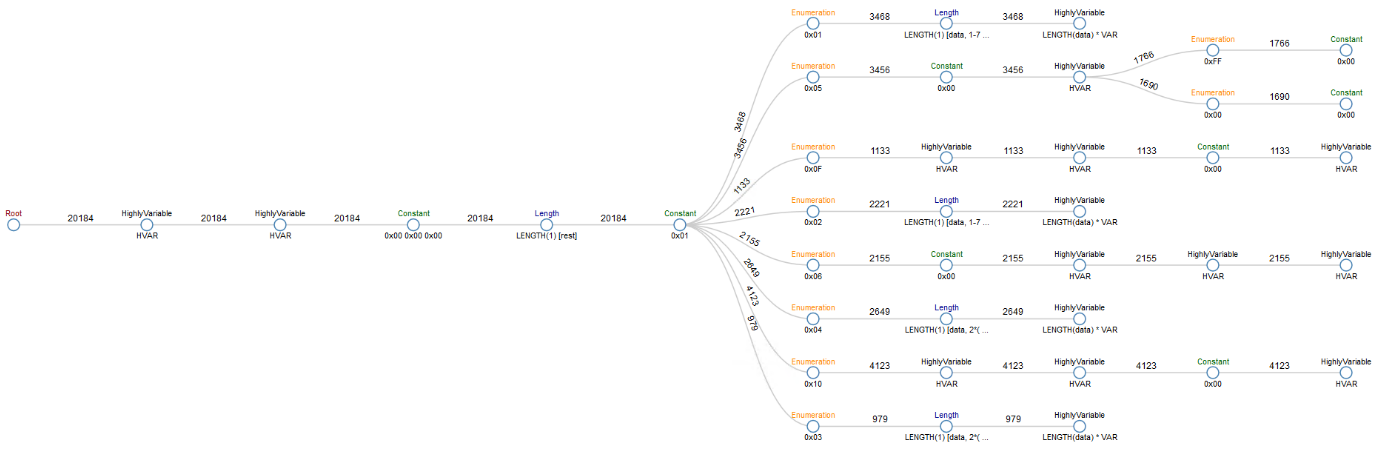

Figure 1. Output of the tree builder algorithm showing the results of a run on a capture of responses of the Modbus protocol.

Append a green leaf to the current branch, advance all pointers by one, then continue processing at 1).

2) Length-prefixed strings - Interpret the next byte as an integer, then test whether this value is followed by this many printable characters. If this test succeeds, a length-prefixed string was found. Append a cyan leaf to the current branch, advance all pointers by one plus the length of the string, then continue processing at 1).
3) Null-terminated strings - Starting from the next byte in each packet, test whether the following bytes can be interpreted as a sequence of printable characters followed by a null byte. If this test succeeds, a null-terminated string was found. Append a cyan leaf to the current branch, advance all pointers past the next null byte, then continue processing at 1).
4) Length fields - Interpret the next four bytes in each packet as a single integer. Test whether this value matches the length of packet (optionally with a given offset). If the test succeeds, these four bytes indicate the length of the packet. Append a blue leaf to the current branch, advance all pointers by four, then continue processing at 1). If the test fails, repeat the same procedure but with the next two bytes only instead of four. If that fails as well, repeat the procedure, this time just with the next single byte.
5) Counters - Interpret the next four bytes in each packet as a single integer. Test whether this value increases by the same amount between packets. If the test succeeds, these four bytes form a counter. Append a purple leaf to the current branch, advance all pointers by four, then continue processing at 1). If the test fails, repeat the same procedure but with the next two bytes only instead of four. If that fails as well, repeat the procedure, this time just with the next single byte.
6) Enumerated types - Check the next byte of each packet. Calculate how many distinct values occur. If this amount is lower than a threshold, we have found an enumerated type. For each distinct value that was seen, append an orange leaf to the current branch, and tag it with one of the previously unused distinct values. Split the list of packets such that each packet is assigned to the branch that is tagged with the value of the packet's next byte. From this point on, only process messages that were assigned to the branch that is currently being processed. Advance all pointers by one. Continue processing at 1) for each of the newly created branches. Since branches are not interdependent, if multiple CPU cores are available, processing may continue in parallel. As for the threshold, based on empirical evidence, values between 8 and 20 seem to be ideal, or if the number of distinct message types is suspected, that number should be used instead.
7) Highly variable - If none of the previous classifiers classified this byte as something else, then it takes on many different values that follow no discernible pattern. Append a black leaf to the current branch, advance all pointers by one, then continue processing at 1).

When no packet on any of the branches has unprocessed bytes left, no more nodes can be added to the tree, and the algorithm ends, outputting the tree. An example of a result can be seen on Figure 1. Note that the colours of the nodes may be arbitrarily chosen as long as each field type is coloured differently.

C. *Optimizations*

Assuming that the protocol being analysed only consists of messages that only contain fields of the previously listed detectable properties, and that the input is of high enough quality (i.e. there are enough messages to analyse on each branch), the tree construction algorithm yields a correct but not necessarily concise result. The resulting tree may be further optimized for one or more metrics, usually at a cost of others.

- Variable length messages - Certain message types, such as write requests with payloads of varying length or responses to read requests will get inferred multiple times: once for each different message length. This

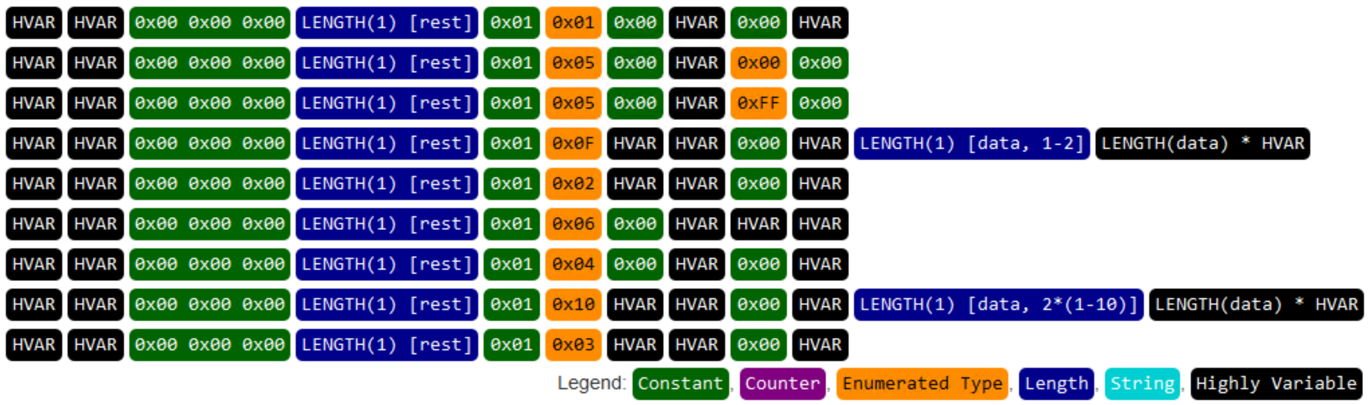
HVAR HVAR 0x00 0x00 0x00 LENGTH(1) [rest] 0x01 0x01 0x00 HVAR 0x00 HVAR
HVAR HVAR 0x00 0x00 0x00 LENGTH(1) [rest] 0x01 0x05 0x00 HVAR 0x00 0x00
HVAR HVAR 0x00 0x00 0x00 LENGTH(1) [rest] 0x01 0x05 0x00 HVAR 0xFF 0x00
HVAR HVAR 0x00 0x00 0x00 LENGTH(1) [rest] 0x01 0x0F HVAR HVAR 0x00 HVAR LENGTH(1) [data, 1-2] LENGTH(data) * HVAR
HVAR HVAR 0x00 0x00 0x00 LENGTH(1) [rest] 0x01 0x02 HVAR HVAR 0x00 HVAR
HVAR HVAR 0x00 0x00 0x00 LENGTH(1) [rest] 0x01 0x06 0x00 HVAR HVAR HVAR
HVAR HVAR 0x00 0x00 0x00 LENGTH(1) [rest] 0x01 0x04 0x00 HVAR 0x00 HVAR
HVAR HVAR 0x00 0x00 0x00 LENGTH(1) [rest] 0x01 0x10 HVAR HVAR 0x00 HVAR LENGTH(1) [data, 2*(1-10)] LENGTH(data) * HVAR
HVAR HVAR 0x00 0x00 0x00 LENGTH(1) [rest] 0x01 0x03 HVAR HVAR 0x00 HVAR
Legend: Constant, Counter, Enumerated Type, Length, String, Highly Variable

Figure 2. Message types of Modbus requests, as read from a graph. Each line represents a unique (detected) message type, with each block denoting a group of bytes (coloured as per the legend). For constants and enumerated types, their values are displayed in the blocks. For length and counter types, their widths and seen value ranges are shown. For everything else, the type of the node is displayed.

phenomenon may be detected by looking for branches that end in a number of highly variable fields that are preceded (not necessarily directly) by a length byte, and are otherwise identical. Message types detected this way may be merged to improve the conciseness score.

- Falsely detected enumerated types - Protocols may contain bytes that contain fields that have a limited range of values (e.g. flags) but don't change the rest of the message structure. These will be inferred as enumerated types, possibly resulting in the same message type(s) getting recognized multiple times. This phenomenon may be detected by looking for identical branches that are preceded by the enumerated type in question. In this case, the branches may be merged and the enumerated type node may be replaced by a brown coloured (Flag) node. This may improve the conciseness score, but may also incorrectly merge truly different message types, resulting in loss of correctness.

D. Interpreting the Results

Once the tree construction is done, and the optional optimization steps are run, the distinct message types may be read from the graph by considering the walks from the root to each leaf node. An example of results can be seen on Figure 2.

IV. Evaluation

The algorithm was evaluated on two commonly used binary protocols, Modbus and MQTT. The goodness of message type inference was measured by the three standard metrics: correctness (1), conciseness (2), and coverage (3), where T is the set of true messages and I is the set of inferred messages.

$$Correctness = \frac{|I \cap T|}{|I|} \tag{1}$$

$$Conciseness = \frac{|I| - |I \setminus T|}{|T| - |T \setminus I|} \tag{2}$$

$$Coverage = \frac{|T \cap I|}{|T|} \tag{3}$$

To measure the accuracy of semantics inference, we defined two metrics: accuracy and adjusted accuracy. Accuracy measures what percentage of field semantics were inferred correctly, while adjusted accuracy considers miscategorizations correct where the miscategorization was a result of the input not being rich enough. For example, consider a two-byte counter that was classified as a one-byte constant followed by a one-byte counter. The accuracy metric considers this incorrect, since this does not strictly match the specification. However, it is considered correct for the adjusted accuracy metric, since this miscategorization was the result of the upper byte never changing values (thus the input not being rich enough).

A. Evaluation with Modbus Traffic

Modbus is a communication protocol originally designed in 1979 for use with PLCs. Today, it is still frequently used with industrial control systems (ICS). Modbus' specification is openly available. Although the specification [20] defines 21 functions (pairs of requests and responses), some of these are only to be implemented for use over serial lines, and a typical implementation only contains 8 of these: 4 kinds of reads and 4 kinds of writes.

For the evaluation, we have recorded approximately 20 000 Modbus request-response pairs on an ICS testbed. This includes Modbus traffic from normal operation as well as several thousands of repeated manual read and write requests with a wide variety of legal parameter values. The source ports of the requests and the destination ports of the responses were edited to be the same with *editcap*, one of the tools from the Wireshark package. This editing was needed to make sure that the packets are recognized to belong to the same message flow. The Modbus payloads were not altered in any manner.

Next, we built models of the Modbus requests and responses based on the true specification. An example of a model is shown on Figure 3). These were then used to calculate the performance metrics for the algorithm (see Table II for results). It can be seen that the algorithm reached maximum correctness and coverage, no matter what optimizations were enabled. Enabling both optimizations also maximized conciseness. The

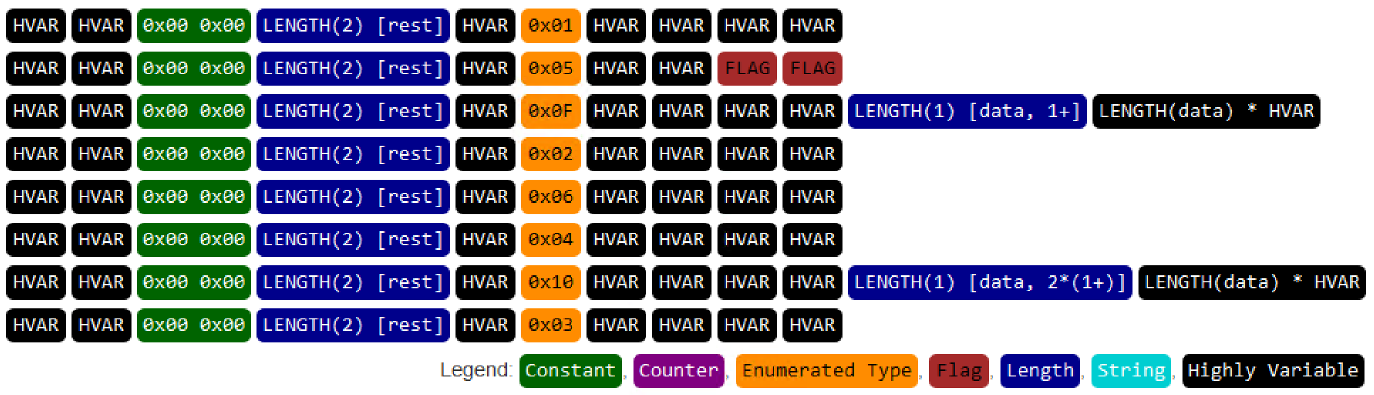
HVAR HVAR 0x00 0x00 LENGTH(2) [rest] HVAR 0x01 HVAR HVAR HVAR HVAR
HVAR HVAR 0x00 0x00 LENGTH(2) [rest] HVAR 0x05 HVAR HVAR FLAG FLAG
HVAR HVAR 0x00 0x00 LENGTH(2) [rest] HVAR 0x0F HVAR HVAR HVAR HVAR LENGTH(1) [data, 1+] LENGTH(data) * HVAR
HVAR HVAR 0x00 0x00 LENGTH(2) [rest] HVAR 0x02 HVAR HVAR HVAR HVAR
HVAR HVAR 0x00 0x00 LENGTH(2) [rest] HVAR 0x06 HVAR HVAR HVAR HVAR
HVAR HVAR 0x00 0x00 LENGTH(2) [rest] HVAR 0x04 HVAR HVAR HVAR HVAR
HVAR HVAR 0x00 0x00 LENGTH(2) [rest] HVAR 0x10 HVAR HVAR HVAR HVAR LENGTH(1) [data, 2*(1+)] LENGTH(data) * HVAR
HVAR HVAR 0x00 0x00 LENGTH(2) [rest] HVAR 0x03 HVAR HVAR HVAR HVAR
Legend: Constant, Counter, Enumerated Type, Flag, Length, String, Highly Variable

Figure 3. A model of Modbus requests, built based on the true specification. To be interpreted in the same way as Figure 2.

TABLE II
PERFORMANCE METRICS OF THE ALGORITHM ON THE MODBUS PROTOCOL

Algorithm	Message Type	Correctness	Conciseness	Coverage	Accuracy	Adjusted Accuracy
Tree construction with no optimizations	Request	1	2.375	1	0.72	0.99
Tree construction with optimization #1	Request	1	1.125	1	0.72	0.99
Tree construction with optimizations #1 and #2	Request	1	1	1	0.73	1
Tree construction with no optimizations	Response	1	4.875	1	0.75	0.9886
Tree construction with optimization #1	Response	1	1.125	1	0.75	0.9886
Tree construction with optimizations #1 and #2	Response	1	1	1	0.7613	1
Tree construction with no optimizations	Average	1	3.625	1	0.735	0.9893
Tree construction with optimization #1	Average	1	1.125	1	0.735	0.9893
Tree construction with optimizations #1 and #2	Average	1	1	1	0.7457	1

differences between accuracy and adjusted accuracy can be explained by the top bytes of length fields and highly variable fields getting detected as constants due to the input packet dump not being of high enough quality.

B. Evaluation with MQTT Traffic

MQTT, or Message Queueing Telemetry Transport is a standard messaging protocol that follows the publish-subscribe pattern. MQTT is fully open, and is typically used in Internet-of-Things (IoT) solutions. The specification defines a total of 14 message types, 5 of which may only be sent by the client, 4 of which may only be sent by the server, and 5 of which may be sent by either party [20].

For the evaluation, we set up an environment with *Eclipse Mosquitto*[1], an open source MQTT server, then used the *HiveMQ Websocket Client*[2] to perform as many operations and with as many different parameter combinations as possible. Traffic was captured on the server using Wireshark, resulting in approximately 1 200 packets. The packets did not need to be altered in any way before analysis.

As with Modbus, we built models based on the true specification, to which we then compared our inferred specification. Results are shown in Table III. Perfect correctness and coverage are achieved in addition to decent conciseness. In the majority of cases, the low (unadjusted) accuracy scores can be attributed to the fact that several messages of the protocol are of fixed length, which results in the algorithm misclassifying length fields as constants.

[1] https://projects.eclipse.org/projects/technology.mosquitto
[2] http://www.hivemq.com/demos/websocket-client/

V. LIMITATIONS AND FUTURE WORK

During evaluation, we have found that the solution presented herein has two limitations that may not be possible to overcome:

- Handling encrypted traffic - Like any other approach that relies on nothing else but network traces, reconstruction fails if the protocol messages are encrypted or are otherwise obfuscated. If the encryption is weak or badly implemented, it may be cracked, or a man-in-the-middle attack may be used against the communicating parties. Failing that, a binary analysis based (or hybrid) approach may still work.
- Poor results for poor inputs - If certain message types were not seen during the capture process, those will be missing from the reconstructed specification, resulting in suboptimal coverage metrics. In addition, if messages for a given type were low in count or variance, then field semantics inference may fail, resulting in low accuracy scores.

We have also identified areas where the solution could be further improved:

- Detection of unicode strings - Currently, only ASCII strings can be detected, but newer protocols may contain messages having unicode strings. We expect that it is possible to detect these strings, however, extensive testing is needed to ensure that this functionality does not introduce false detections.
- Split-byte fields - Some protocols, including MQTT, don't always use whole bytes to store information (e.g. the top

TABLE III
PERFORMANCE METRICS OF THE ALGORITHM ON THE MQTT PROTOCOL

Algorithm	Message Type	Correctness	Conciseness	Coverage	Accuracy	Adjusted Accuracy
Tree construction (any optimization settings)	Client	1	1.2	1	0.5483	0.9677
Tree construction without optimization #2	Server	1	1	1	0.7333	1
Tree construction with optimization #2	Server	1	1	1	0.8	1
Tree construction without optimization #2	Shared	1	2	1	0.7391	0.9565
Tree construction with optimization #2	Shared	1	1	1	0.7391	0.9565
Tree construction without optimization #2	Average	1	1.4	1	0.6735	0.9747
Tree construction with optimization #2	Average	1	1.06	1	0.6958	0.9747

four bits of a byte might be flags, while the lower four could be a counter). The algorithm could be reworked to try to detect and handle these cases.

- Model merging - It should be possible to merge two inferred models of the same protocol in order to improve the resulting specification. The details still need to be worked out.
- Leaving room for error - It is currently assumed that no packets are lost, duplicated or corrupted during transmission and capture. One of these events occurring may result in most types not being detected correctly. This issue could be worked around by allowing a small amount of corrupted or out-of-sequence packets. However, this could also result in false detections, thus should be a subject of further research.

With these improvements done, it would be possible to generate protocol specifications that are accurate enough to be used directly as a basis of fuzz testing, honeypots or firewall rules, among others. Furthermore, we plan to investigate how the results of the tree building algorithm could be used as inputs to other algorithms that aim to infer protocol grammar or otherwise try to find correlations between fields in requests and responses.

VI. CONCLUSION

In this paper, we have presented a novel method to infer message types and field semantics for binary protocols. Our method relies exclusively on network traces, and works by constructing and optimizing an acyclic graph based on the contents of the packets in the trace. We have presented a methodology to evaluate the performance of the algorithm, then performed evaluations against the known specifications of two commonly used protocols. Based on the results, we conclude that the approach is already viable, but may be further improved in the future.

REFERENCES

[1] H. J. Wang, C. Guo, D. R. Simon, and A. Zugenmaier, ”Shield: Vulnerability-Driven Network Filters for Preventing Known Vulnerability Exploits,” Proceedings of the ACM SIGCOMM 2004 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication, pp. 193–204, 2004.

[2] T. Krueger, H. Gascon, N. Krämer, and K. Rieck, ”Learning Stateful Models for Network Honeypots,” Proceedings of the 5th ACM Workshop on Artificial Intelligence and Security, pp. 37–48, 2012.

[3] J. Antunes, N. Neves, M. Correia, P. Verissimo, and R. Neves, ”Vulnerability Discovery with Attack Injection,” IEEE Transactions on Software Engineering, vol. 36, no. 3, pp. 357-370, 2010.

[4] J. Narayan, S. K. Shukla, and T. C. Clancy, ”A Survey of Automatic Protocol Reverse Engineering Tools,” ACM Computing Surveys, vol. 48, no. 3, art. 40, 2016.

[5] J. Duchêne, C. Le Guernic, E. Alata, V. Nicomette, and M. Kaâniche, ”State of the art of network protocol reverse engineering tools,” Journal of Computer Virology and Hacking Techniques, vol. 14, no. 1, pp. 53–68, 2018.

[6] D. Lee and M. Yannakakis, ”Principles and Methods of Testing Finite State Machines —- A Survey,” Proceedings of the IEEE, vol. 84, no. 8, pp. 1090–1123, 1996.

[7] M. A. Beddoe, ”Network Protocol Analysis using Bioinformatics Algorithms,” http://www.4tphi.net/˜awalters/PI/PI.html, 2004.

[8] W. Cui, J. Kannan, and H. J. Wang, ”Discoverer: Automatic Protocol Reverse Engineering from Network Traces,” SS’07 Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium, art. 14, 2007.

[9] Y. Wang, X. Li, J. Meng., Y. Zhao, Z. Zhang, and L. Guo, ”Biprominer: Automatic Mining of Binary Protocol Features,” 12th International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT), pp. 179–184, 2011.

[10] J. Antunes, N. Ferreira, and P. Verissimo, ”ReverX: Reverse Engineering of Protocols,” Docs.DI, the repository of the Department of Informatics of the University of Lisbon, Faculty of Sciences, 2011.

[11] Y. Wang, X. Yun, M. Z. Shafiq, L. Wang, A. X. Liu, et al., ”A semantics aware approach to automated reverse engineering unknown protocols,” 20th IEEE International Conference on Network Protocols (ICNP), pp. 1–10, 2012.

[12] J-Z. Luo, and S-Z. Yu, ”Position-based automatic reverse engineering of network protocols,” Journal of Network and Computer Applications, vol. 36, no. 3, pp. 1070–1077, 2013.

[13] J. Caballero, H. Yin, Z. Liang, and D. Song, ”Polyglot: Automatic Extraction of Protocol Message Format using Dynamic Binary Analysis,” CCS ’07 Proceedings of the 14th ACM conference on Computer and communications security, pp. 317–329, 2007.

[14] Z. Lin, X. Jiang, D. Xu, and X. Zhang, ”Automatic Protocol Format Reverse Engineering through Context-Aware Monitored Execution,” 15th Symposium on Network and Distributed System Security (NDSS), 2008.

[15] Z. Wang, X. Jiang, W. Cui, X. Wang, and M. Grace, ”ReFormat: Automatic Reverse Engineering of Encrypted Messages,” Proceedings of the 14th European Symposium on Research in Computer Security (ESORICS), pp. 200–215, 2009.

[16] C. Leita, K. Mermoud, and M. Dacier, ”ScriptGen: an automated script generation tool for Honeyd,” 21st Annual Computer Security Applications Conference, pp. 203–214, 2005.

[17] P. M. Comparetti, G. Wondracek, C. Kruegel, and E. Kirda, ”Prospex: Protocol Specification Extraction,” 30th IEEE Symposium on Security and Privacy, pp. 110–125, 2009.

[18] Y. Wang, Z. Zhang, D. Yao, B. Qu, and L. Guo, ”Inferring Protocol State Machine from Network Traces: A Probabilistic Approach,” ACNS 2011: Applied Cryptography and Network Security, pp. 1–18, 2011.

[19] C. Y. Cho, D. Babić, P. Poosankam, K. Z. Chen, E. X. Wu, and D. Song, ”MACE: Model-inference-Assisted Concolic Exploration for Protocol and Vulnerability Discovery,” SEC’11 Proceedings of the 20th USENIX conference on Security, art. 10, 2011.

[20] Modbus Organization, Inc., ”Modbus Application Protocol Specification V1.1b3”, 2012.

[21] A. Banks, R. Gupta, ”MQTT Version 3.1.1 (OASIS Standard)”, 2014, http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/mqtt-v3.1.1.html.