GrAMeFFSI: Graph Analysis Based Message Format and Field Semantics Inference For Binary Protocols, Using Recorded Network Traffic

Gergő Ládi[1], Levente Buttyán[2], and Tamás Holczer[3]

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 relying solely on recorded network traffic. More specifically, we propose GrAMeFFSI, a method based on graph analysis 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 official specifications of these protocols.**

Index Terms—**protocol reverse engineering, message format, field semantics, inference, binary protocols, network traffic, graph analysis, 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. 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), however, we do not aim to reconstruct the state machine in this paper.

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 message types were found.

Based on how messages are represented, 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-

[1,2,3] Laboratory of Cryptography and System Security, Department of Networked Systems and Services, Budapest University of Technology and Economics, Budapest, Hungary

[1] BME Balatonfüred Student Research Group, Hungary

E-mail: {gergo.ladi, buttyan, holczer}@crysys.hu

DOI: 10.36244/ICJ.2020.2.4

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 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 GrAMeFFSI, a novel graph analysis based algorithm for binary protocols which can infer not only the message types, but also a variety of field semantics, using only network traces of the protocols. We implement and test 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. In addition, we introduce two metrics, accuracy and adjusted accuracy, to measure the goodness of semantics inference. We also show that GrAMeFFSI can infer field semantics with over 95% accuracy if high quality network traces are available.

This paper revises, improves, and extends our previous work, *Message Format and Field Semantics Inference for Binary Protocols Using Recorded Network Traffic* [6]. Notable additions are a model merging phase in the algorithm and the mathematical formalization of the metrics. The model merging phase further improves the accuracy of our algorithm while also providing extra semantical information, and the formalization aims to make our results possible to reproduce as well as make it easier to compare it to other works (where such metrics are used).

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 [7]. The first well-known project that aimed at restoring the specifications of a computer protocol was the Protocol Informatics Project by M. A. Beddoe [8] in 2004, which used bioinformatical algorithms such as the well-known Needleman-Wunsch sequence alignment algorithm on network traces to infer the message types of the text-based protocol HTTP. It was later followed by Discoverer [9], Biprominer [10], ReverX [11], ProDecoder [12], and AutoReEngine [13] that all relied only on network traffic. While most algorithms aimed at reversing both text-based and binary protocols, some specialized in one or the other, typically achieving better performance metrics compared to the more general solutions of their time. Biprominer, as its name suggests, targeted binary protocols, while ReverX targeted text-based protocols. The methods employed vary – Discoverer relies on sequence alignment, Biprominer and AutoReEngine leverage data mining approaches, while ProDecoder makes use of natural language processing algorithms.

Early works typically focused on reverse engineering the message formats and their syntax, and did not put much emphasis on inferring field semantics (that is, what each of the fields means). Even those that tried did not achieve significant results – Discoverer admits to achieving between 30-40% accuracy [9], and not even Netzob exceeds 50% [14]. FieldHunter [15] from 2015 was the first to achieve over 80% accuracy on semantics.

Methods relying on reversing implementations appeared under the names of Polyglot [16], AutoFormat [17], and ReFormat [18]. These generally work on the principles of dynamic taint analysis, marking pieces of code in the memory area of a running executable that are run in response to a given message, then making assumptions about the message formats based on what and how was marked. It has been proven [4] that binary analysis based approaches can achieve better results, however, purely traffic analysis based approaches are also important as binaries may not always be at our disposal and legal agreements may prevent us from analysing or reverse engineering these.

Solutions to reverse the protocol grammar (the state machine of the protocol) have also been proposed in the form of ScriptGen [19], Prospex [20], Veritas [21], and MACE [22]. However, they are not in scope of this paper as we currently do not aim to reconstruct the state machine of the protocol.

In this paper, we aim to compete with Discoverer, Biprominer, and ProDecoder, three different approaches for reversing the message formats of binary protocols; as well as Netzob and FieldHunter that aim at extracting semantic information. The performance statistics of these solutions, as given by their authors (or calculated based on their respective papers), are shown in Table I.

We believe that no prior protocol message format reversing method exists that is based on graph operations.

III. Our Approach

Our approach consists of five 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 directed acyclic connected graphs (rooted trees) based on the input. Next, in the third phase, we merge the trees from phase two, following a set of rules. In the fourth phase, (optional) optimizations may be run on the trees. These optimizations generally improve a certain metric at a possible cost of impairing a different metric. Finally, the resulting tree is used to enumerate the inferred message types and field semantics.

TABLE I
PERFORMANCE METRICS OF SIMILAR APPROACHES

Approach	Correctness	Conciseness	Coverage	Accuracy	Tested on # protocols
Discoverer	0.9	5	0.95	30-40%	3
Biprominer	0.99	Unknown	0.967	N/A	3
ProDecoder	0.975	Unknown	0.975	N/A	2
Netzob	0.775	1.74	Unknown	33.4%	4
FieldHunter	Unknown	2.1	Unknown	91.89%	7

Notes: Values for Netzob are only approximately accurate as they were manually read from a plot. For FieldHunter, only binary protocols were considered.

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 or connecting them via a hub (or a switch with port mirroring configured), and then capturing traffic from a third device that is also connected to the hub.

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.

It is highly preferable to repeat the traffic generation procedure a couple of times, disconnecting and reconnecting the client and the server (or the peers) in between. This ensures that multiple flows (sessions, connections) are recorded. Since certain values such as session identifiers never change during a single session, recording multiples of them is necessary in order to achieve more accurate results. Similarly, if multiple clients and servers (or peers) are available, it is also imperative to record at least one full session in each possible valid combination thereof. This ensures that fields containing identifiers that are unique and never change for each client (e.g. factory-set device IDs) can still be detected as such.

B. Tree Construction

In the second phase, the recorded traffic is processed and a tree is constructed for each flow based on the messages that appear in that given flow. These trees represent the suspected message types and field semantics as deduced from the data seen, and will be further processed in later steps.

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. 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

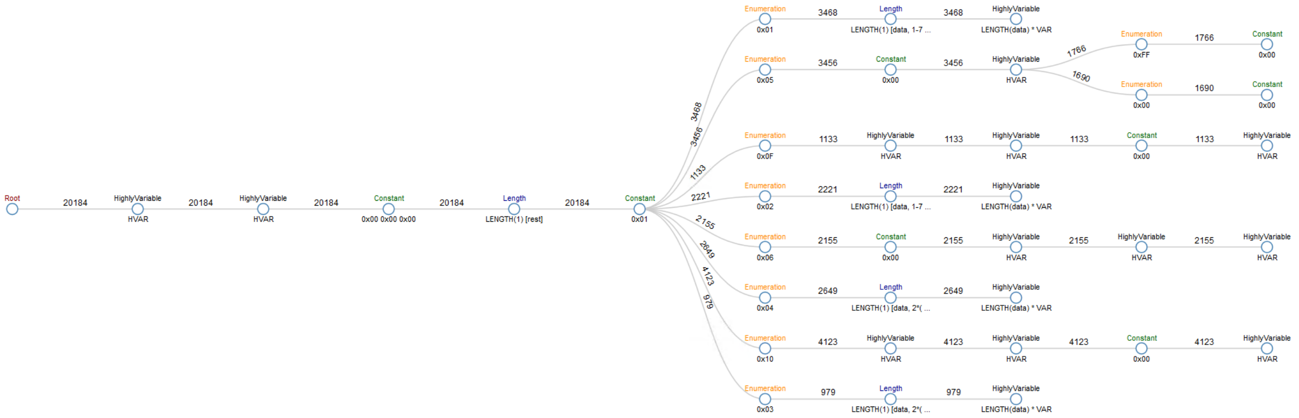


Figure 1. Output of the tree builder algorithm showing the results of a run on a capture of responses of the Modbus protocol.

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).

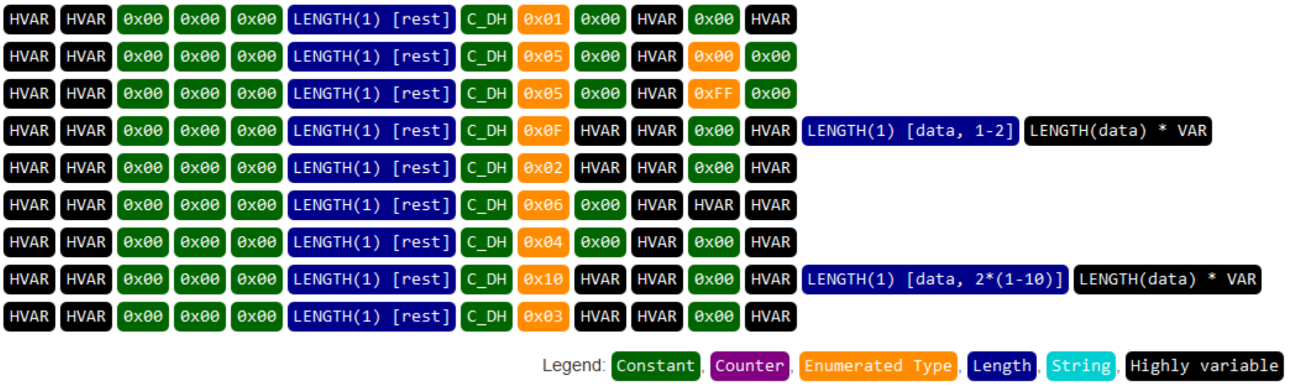
When no packet on any of the branches has unprocessed bytes left, no more nodes can be added to the tree, and the algorithm ends, outputting the tree. An example of a result can be seen on Figure 1. Note that the colours of the nodes may be arbitrarily chosen as long as each field type is coloured differently.

C. Model Merging

If we just considered each flow individually, it would not be possible to find mutually exclusive message types (as at least one of these would be missing in each flow), and it would also not be possible to find fields containing session identifiers (as these would appear constant within each flow). However, merging the trees and correlating data from the previous step solves such issues, greatly improving the resulting inferred specification if the right network traces are available.

The merging process is as follows: starting from the root node, compare the next child node of each tree using the following rules:

1) If all are of type *counter*, *flag*, *length*, *string* or *variable*, continue merging the direct descendants.
2) If all are of type *enumerated*, continue merging each subtree where the value of the *enumerated* node is the same (this may be parallelized). If a value only appears once, add it with all of its children to the resulting tree. Alternatively, if this step results in too many (exact value varies on a case-by-case basis) branches, this may be the case where a *variable* type gets detected as an *enumerated type* due to the inputs being poor – in this case, the *enumerated type* may be replaced with a *variable* type, and all subtrees may be merged into one.
3) If all are of type *constant*, check the value of the nodes. If the value is always the same, it's a generic constant. If the value is always the same for the same client (or server) and is different for other clients (or servers), it's a source or destination host identifier. If the value is only the same within each flow, then it must be a session identifier.
4) If some are of type *constant* and all others are of the same (non-constant) type, proceed as if everything was of that other non-constant type.
   For example, suppose we have five trees. The next element is a *length* field according to three of these, and it's a *constant* according to the other two. The field should be treated as if it was a *length* field in all five trees.
5) Any other combinations are rare and typically indicate a problem with the recorded traffic or the implementations themselves. In such cases, the field should be treated as if it was of type *variable* in all of the trees.



HVAR	HVAR	0x00	0x00	0x00	LENGTH(1) [rest]	C_DH	0x01	0x00	HVAR	0x00	HVAR		
HVAR	HVAR	0x00	0x00	0x00	LENGTH(1) [rest]	C_DH	0x05	0x00	HVAR	0x00	0x00		
HVAR	HVAR	0x00	0x00	0x00	LENGTH(1) [rest]	C_DH	0x05	0x00	HVAR	0xFF	0x00		
HVAR	HVAR	0x00	0x00	0x00	LENGTH(1) [rest]	C_DH	0x0F	HVAR	HVAR	0x00	HVAR	LENGTH(1) [data, 1-2]	LENGTH(data) * VAR
HVAR	HVAR	0x00	0x00	0x00	LENGTH(1) [rest]	C_DH	0x02	HVAR	HVAR	0x00	HVAR		
HVAR	HVAR	0x00	0x00	0x00	LENGTH(1) [rest]	C_DH	0x06	0x00	HVAR	HVAR	HVAR		
HVAR	HVAR	0x00	0x00	0x00	LENGTH(1) [rest]	C_DH	0x04	0x00	HVAR	0x00	HVAR		
HVAR	HVAR	0x00	0x00	0x00	LENGTH(1) [rest]	C_DH	0x10	HVAR	HVAR	0x00	HVAR	LENGTH(1) [data, 2*(1-10)]	LENGTH(data) * VAR
HVAR	HVAR	0x00	0x00	0x00	LENGTH(1) [rest]	C_DH	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.

For example, an implementation might generate request identifiers sequentially, while others might choose them randomly. In this case, the field containing the request identifier will be recognized as a *counter* for the former implementations, while it will be recognized as *variable* for the latter ones.

D. 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 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.

E. 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 goodness of message type inference was measured by the three standard metrics, correctness (1), conciseness (2), and coverage (3):

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

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

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

where T is the set of true messages and I is the set of inferred messages.

To calculate these three metrics, we need the true and the inferred models of the message types, as well as a network capture that contains each true message type at least once. Then, the following algorithm can be used:

1) Initialization: Begin with an empty list of mappings, M_{TI}, which will contain mappings from true message types to inferred message types.
2) Mapping creation: For each protocol message that exists in the network capture: find out which message type it corresponds to in the sets of true and inferred message types. If it matched something in both sets, say, T_x among the true message types and I_y among the inferred message types, then add a $T_x \mapsto I_y$ mapping to M_{TI}. (If the exact same mapping is already on the list, it should not be added a second time.)

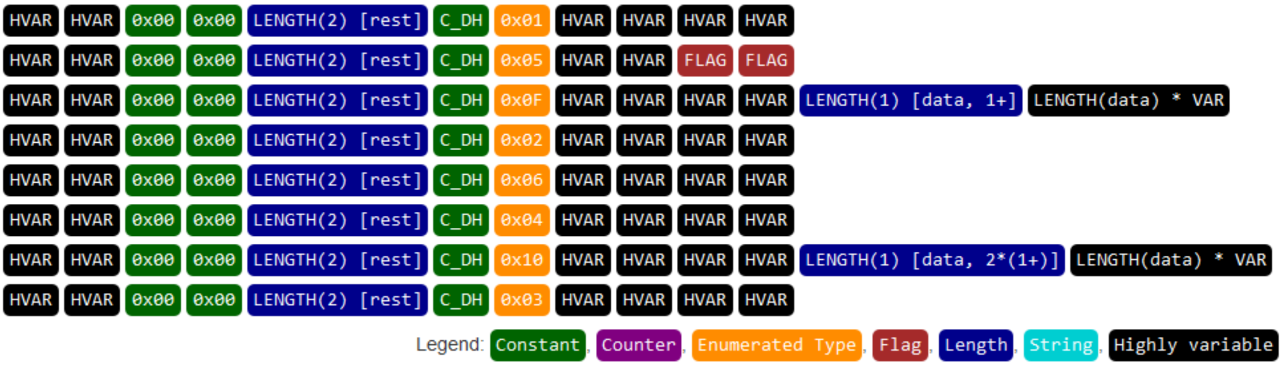

HVAR HVAR 0x00 0x00 LENGTH(2) [rest] C_DH 0x01 HVAR HVAR HVAR HVAR
HVAR HVAR 0x00 0x00 LENGTH(2) [rest] C_DH 0x05 HVAR HVAR FLAG FLAG
HVAR HVAR 0x00 0x00 LENGTH(2) [rest] C_DH 0x0F HVAR HVAR HVAR HVAR LENGTH(1) [data, 1+] LENGTH(data) * VAR
HVAR HVAR 0x00 0x00 LENGTH(2) [rest] C_DH 0x02 HVAR HVAR HVAR HVAR
HVAR HVAR 0x00 0x00 LENGTH(2) [rest] C_DH 0x06 HVAR HVAR HVAR HVAR
HVAR HVAR 0x00 0x00 LENGTH(2) [rest] C_DH 0x04 HVAR HVAR HVAR HVAR
HVAR HVAR 0x00 0x00 LENGTH(2) [rest] C_DH 0x10 HVAR HVAR HVAR HVAR LENGTH(1) [data, 2*(1+)] LENGTH(data) * VAR
HVAR HVAR 0x00 0x00 LENGTH(2) [rest] C_DH 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.

3) Correctness: count the number of distinct T_is that appear on the left-hand side in mappings in M_{TI} – this is the number of correctly inferred message types. Count the number of I_js that never appear on the right-hand side in mappings in M_{TI} – this is the number of bogus (inferred but nonexistent) messages types. Finally, to get the correctness, divide the number of correctly inferred types by the sum of correctly inferred and bogus message types.
4) Conciseness: subtract the number of bogus message types from the total number of inferred message types. Divide this number by the number of true message types minus the number of message types that were not found. The number of message types that were not found can be calculated by counting the number of T_is that never appear on the left-hand side in mappings in M_{TI}.
5) Coverage: Divide the number of correctly inferred message types by the number of elements in T.

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 accepts miscategorized bytes as 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).

To compute the accuracy and adjusted accuracy scores, we use a Tree Edit Distance (TED) algorithm. The TED is a measure of how similar two trees are. It is generally defined as the minimum cost sequence of edit operations that transforms one tree into the other (4) [23].

$$TED(t_1,t_2) = \min_{(e_1,\ldots,e_k)\in\mathcal{P}(t_1,t_2)} \sum_{i=1}^{k} c(e_i) \qquad (4)$$

We have chosen APTED [24, 25], one of the state-of-the-art TED algorithms. It supports three types of edit costs (weights): node insertion, node deletion, and node renaming (relabeling). It has a Java-based implementation available[4] on Github, which we ported to C# and published[5] on Github. Running APTED on the graphs of the true and the inferred specifications with the weights $(0, 0, 1)$ for insertion, deletion, and relabeling respectively, we get the number of bytes that were incorrectly inferred semantically. Subtracting this number from the total number of bytes in the graph, then dividing the result by the total yields the accuracy. Using 0 as weights for insertion and deletion ensures that bogus and duplicate messages, as well as ones that were not found are not considered when calculating the accuracy of semantics inference. Adjusted accuracy is calculated similarly, by using $(0, 0, f(n_1, n_2))$ as weights, where f returns 0 not just when the labels of n_1 and n_2 are equal, but also when the inferred node is constant; in any other cases, f returns 1.

GrAMeFFSI was evaluated on two commonly used binary protocols, Modbus and MQTT.

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 [26] 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, nor were the IP addresses that are used to determine which device is which for host identifier inference.

[4] https://github.com/DatabaseGroup/apted
[5] https://github.com/GergoLadi/APTEDSharp/

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	0.8	0.99
Tree construction with optimization #1	Request	1	1.125	1	0.8	0.99
Tree construction with optimizations #1 and #2	Request	1	1	1	0.81	1
Tree construction with no optimizations	Response	1	4.875	1	0.8409	0.9886
Tree construction with optimization #1	Response	1	1.125	1	0.8409	0.9886
Tree construction with optimizations #1 and #2	Response	1	1	1	0.8523	1
Tree construction with no optimizations	Average	1	3.625	1	0.8205	0.9893
Tree construction with optimization #1	Average	1	1.125	1	0.8205	0.9893
Tree construction with optimizations #1 and #2	Average	1	1	1	0.8312	1

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 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 [27].

For the evaluation, we set up an environment with *Eclipse Mosquitto*[5], an open source MQTT server, then used the *HiveMQ Websocket Client*[6] 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 GrAMeFFSI misclassifying length fields as constants.

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 GrAMeFFSI 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 upper 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.
- 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.

[6] https://projects.eclipse.org/projects/technology.mosquitto

[7] http://www.hivemq.com/demos/websocket-client/

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

VI. CONCLUSION

In this paper, we have presented GrAMeFFSI, a novel method to infer message types and field semantics for binary protocols. Our method relies exclusively on network traces, and works by constructing, merging, and optimizing acyclic graphs 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 surpasses existing similar solutions in terms of correctness, conciseness and coverage, while also providing more accurate field semantics in most of the cases.

ACKNOWLEDGMENT

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, Thematic Fundamental Research Collaborations Grounding Innovation in Informatics and Infocommunications).

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. DOI: 10.1145/1015467.1015489

[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. DOI: 10.1145/2381896.2381904

[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. DOI: 10.1109/TSE.2009.91

[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, 2016. DOI: 10.1145/2840724

[5] J. Duchêne, C. L. 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. DOI: 10.1007/s11416-016-0289-8

[6] G. Ládi, L. Buttyán, and T. Holczer, “Message format and field semantics inference for binary protocols using recorded network traffic,” *26th International Conference on Software, Telecommunications and Computer Networks*, 2018. DOI: 10.23919/SOFTCOM.2018.8555813

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

[8] M. A. Beddoe, “Network protocol analysis using bioinformatics algorithms,” http://www.4tphi.net/32awalters/PI/PI.html, 2004.

[9] 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*, 2007.

[10] 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. DOI: 10.1109/PDCAT.2011.25

[11] J. Antunes, N. Ferreira, and P. Verissimo, “ReverX: Reverse engineering of protocols,” *12th International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT)*, 2011.

[12] 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. DOI: 10.1109/ICNP.2012.6459963

[13] 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. DOI: 10.1016/j.jnca.2013.01.013

[14] G. Bossert, F. Guihéry, and G. Hiet, “Towards automated protocol reverse engineering using semantic information,” *Proceedings of the 9th ACM symposium on Information, computer and communications security*, pp. 51–62, 2014. DOI: 10.1145/2590296.2590346

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

[16] 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. DOI: 10.1145/1315245.1315286

[17] 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.

[18] 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. DOI: 10.1007/978-3-642-04444-1_13

[19] 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. DOI: 10.1109/CSAC.2005.49

[20] 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. DOI: 10.1109/SP.2009.14

[21] 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. DOI: 10.1007/978-3-642-21554-4_1

[22] 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*, 2011.

[23] X. Gao, B. Xiao, D. Tao, and X. Li, "A survey of graph edit distance," *Pattern Analysis and Applications*, vol. 13, no. 1, pp. 113–129, 2010. DOI: 10.1007/s10044-008-0141-y

[24] M. Pawlik and N. Augsten, "Efficient computation of the tree edit distance," *ACM Transactions on Database Systems (TODS)*, vol. 40, no. 1, 2015. DOI: 10.1145/2699485

[25] ——, "Tree edit distance: Robust and memory-efficient," *Information Systems*, vol. 56, pp. 157–173, 2016. DOI: 10.1016/j.is.2015.08.004

[26] Modbus Organization, Inc., "Modbus application protocol specification v1.1b3," http://www.modbus.org/docs/Modbus Application Protocol V1 1b3.pdf, 2012.

[27] A. Banks and R. Gupta, "MQTT Version 3.1.1 (OASIS Standard)," http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/mqttv3.1.1. html, 2014.

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

Levente Buttyán received the M.Sc. degree in Computer Science from the Budapest University of Technology and Economics (BME) in 1995, and earned the Ph.D. degree from the Swiss Federal Institute of Technology – Lausanne (EPFL) in 2002. In 2003, he joined the Department of Networked Systems and Services at BME, where he currently holds a position as an Associate Professor and leads the Laboratory of Cryptography and Systems Security (CrySyS Lab). He has done research on the design and analysis of secure protocols and privacy enhancing mechanisms for wireless networked embedded systems (including wireless sensor networks, mesh networks, vehicular communications, and RFID systems). He was also involved in the analysis of some high profile targeted malware, such as Duqu, Flame, MiniDuke, and TeamSpy. His current research interest is in security of cyber-physical systems (including industrial automation and control systems, modern vehicles, cooperative intelligent transport systems, and the Internet of Things in general). Levente Buttyán played instrumental roles in various national and international research projects, published 150+ refereed journal articles and conference/workshop papers, and co-authored multiple books and patents. Besides research, he teaches courses on applied cryptography and IT security at BME and at the Aquincum Institute of Technology (AIT Budapest), and he leads a talent management program in IT security in the CrySyS Lab. He also co-founded multiple spin-off companies, notably Tresorit, Ukatemi Technologies, and Avatao.

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