Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Networked Systems and Services

Gergő Ládi, Tamás Holczer

Report for the Research Scholarship AUTOMATED PROTOCOL REVERSE ENGINEERING

{gergo.ladi, holczer}@crysys.hu

BUDAPEST, 2017

Abstract

In recent years, the automated analysis of unknown communication protocols has become a topic of great interest. With the multitudes of new Industrial Control Systems and Internet of Things devices came several under- or undocumented protocols that need to be understood in order to develop firewalls, honeypots, intrusion detection systems, as well as to test the various implementations for vulnerabilities. The aim of this research was to review existing literature, understand the typical approaches, then design and implement a prototype or a protocol analysis tool that is capable of determining the message types used by a given protocol, based on captured network packets.

1 Introduction

The idea of needing and having automated tools for protocol reverse engineering is not new, solutions have been under research since the early 2000s. The aim of automated protocol reverse engineering (or APRE for short) is to attempt to analyze and describe a (partially) unknown protocol (i.e. the language that is being spoken ant its rules) that is used by a given network application or device, all with as little human interaction as possible. APRE approaches may be used to analyze and understand closed protocols where the protocol documentation is not available. APRE algorithms aid us by generating syntax graphs that describe the messages of the protocol, building finite state machines (FSMs) that describe the possible sequences of messages, or both. Sequence graphs explain the structure of the messages – what data types are represented by each of the bytes (e.g. strings, integers, bitfields, etc.), as well as what the possible values are. FSMs show how and in what order the messages can be sent. Having semi-automatically generated protocol descriptions can help us learn how certain malware communicate with their command-and-control (C&C) servers, or how the proprietary protocols employed by the various industrial control systems (ICS) work. In addition, these protocol descriptions can be used to build intrusion detection/prevention systems (IDS/IPS), or fuzz a concrete implementation for vulnerabilities. Given that many new protocols surface each year and that analysis by hand is rather time consuming, manual analysis is no longer feasible. There are two general approaches to automatic reversing: binary analysis and network trace analysis. The former relies on having an executable that has an implementation of the protocol, then it either tries to examine the code statically (without running it) or dynamically – running the code, looking at which instructions are being executed and in what order when interpreting messages. The latter relies on having nothing but as many messages as possible in the form of a network capture (typically, a Wireshark .pcap file). For this project, we have chosen the way of network trace analysis for two main reasons: one, we expect not to have a binary for most of the tasks to analyze; two, the disassembly and/or analysis of certain binaries may be disallowed by their license agreements.

2 Related Work

This section aims to provide an overview of the state-of-the-art approaches. Since we do not employ binary analysis for this project, approaches related to binary analysis will not be mentioned.

Over the years, there have been many attempts to engineer algorithms that have good correctness, conciseness, and coverage metrics, algorithms that can process various different kinds of protocols. Correctness measures how closely the reverse engineered specification matches the true specification, conciseness measures how many reversed messages represent a single message of the true specification (on average), while coverage measures how many of the true messages were detected. There is an evident distinction among algorithms that aim to reverse binary protocols (such as SMB) and the ones that aim to reverse text-based protocols (such as SMTP).

2.1 Reversing Protocol Syntax

One of the earliest automated approaches is named *Protocol Informatics* (Beddoe, 2004), which was written for and tested on HTTP and ICMP. The algorithm uses message alignment (Needleman-Wunsch) and keyword detection (UPGMAA clustering). *Discoverer* (Cui et al., 2007) has a three-phased approach: first, it attempts to tokenize the messages, then it uses a recursive clustering algorithm to find distinctive tokens; finally, it merges similar clusters. The next approaches, *Biprominer* and *ProDecoder* (Wang et al., 2011 and 2012) focus on binary protocols, and work by finding patterns, grouping these into distinguishing keywords, followed by a probable message sequence generation phase. *ProDecoder* is essentially an improved version of *Biprominer* that also uses the Needleman-Wunsch sequence alignment. *ReverX* (Antunes et al., 2011) uses a speech recognition algorithm to identify delimiters in a text-only protocol, which are then utilized to find keywords based on the frequency of certain byte sequences. Finally, *AutoReEngine* (Luo et al., 2013) first identifies keywords, orders them into vectors, then classifies these into message types using positional variance.

Based on the above, it can be said that most current approaches use some sort of tokenization and keyword detection, followed by a grouping algorithm, then by a clean-up merge phase.

2.2 Reversing the Finite State Machine

In order to build the FSM for a protocol, the syntax has to be known, meaning that FSM reversal algorithms often rely on the results of the previously introduced syntax reversing tools.

The first two tools were *ScriptGen* (Leita et al., 2005) and *RolePlayer* (Cui et al., 2006), which did not actually aim to reverse the FSM, but instead, to emulate command-and-control messages for the analysis of botnets. These observed and stored all messages, made clusters out of them, found edges among the clusters, then built lists of previously observed messages, which were then used to send replies to incoming messages. The first approach to generate true FSMs was an algorithm by Shevertalov et al. (2007), which focused on text-based protocols, and worked by clustering similar messages based on a distance metric, building a graph of states (based on the message contents), removing not-so-frequent states, then combining states with similar message types. The previously-mentioned *ReverX* (Antunes et al., 2011) can also reverse the protocol FSM, and works in a similar fashion. Finally, *Veritas* (Wang et al., 2011) extends these previous ideas by adding a probabilistic model using a kind of a Kolmogorov-Smirnov test. This increases the accuracy by filtering states that have a high probability of appearance, but are not essential to determine the message type.

In general, we can say that the most notable difference is that while syntax reversing tools attempt to identify and cluster data within the messages, FSM reversing tools aim to do the same, but with the messages themselves. Each algorithm uses some kind of message clustering, creates states based on the clusters, filters states, merges similar states, then simplifies the graph.

3 Automated Protocol Reverse Engineering Using Graph Analysis

3.1 Our Approach

Based on what the implementation will be used for, our approach was to combine the some of the previous ideas in order to build an algorithm that can reverse engineer different (unknown) protocols relatively effectively. We don’t need to have a perfectly automatic algorithm; a certain degree of human interaction is allowed. This is expected to reduce the overall complexity and increase completeness metrics. For now, we’re working with binary protocols, so it is safe to assume that the protocols to be reversed are left-to-right (meaning that the messages are read and interpreted from the left side to the right side, sequentially), like most other binary protocols.

The algorithm starts with a graph that has a single vertex, a root node, and one (huge) list of network packets that contain the messages to be analysed. As the first step, the algorithm reads the first byte of each packet, performs various checks and statistical analysis, then creates one or more new vertices based on these results, then these are connected to the previous one. If only one vertex was created, analysis proceeds with the next byte of each message. Otherwise, the list is split in as many parts as many new vertices were created, and processing continues on each branch separately.

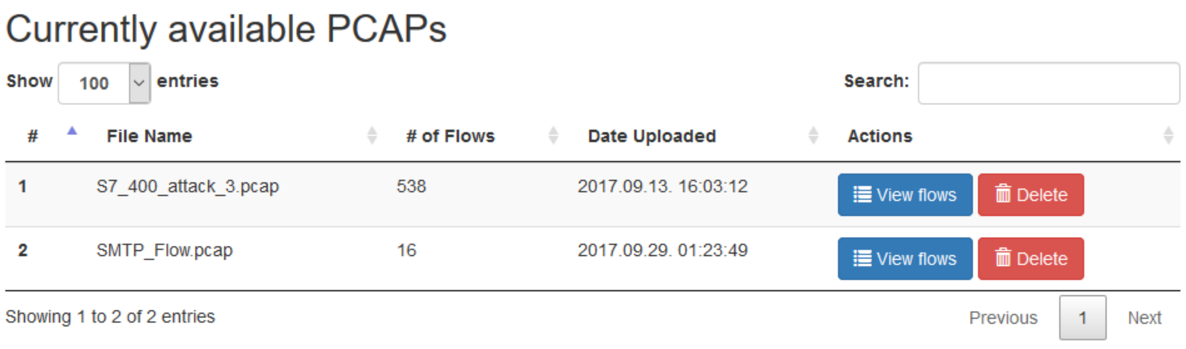
For example, if the first byte was 0x42 for every single message, a vertex having a value of 0x42 will be added to the graph, then connected to the root. Then, if the second byte had two distinct values, say 0x10 and 0x20, two new vertices will be created and connected to the previous one (0x42). The message list is split such that messages that had 0x10 as their second byte will be associated with the vertex 0x10, and messages that had 0x20 as their second byte will be associated with the vertex 0x20.

Following this strategy, it is easy to detect most messages and most data types if enough packets are available. If a byte is always the same for all messages, it is a constant. If a single, double, or quad-byte value always keeps incrementing by the same amount in each packet, it is a counter. If a single, double, or quad-byte value is always equal to the length of the packet (plus maybe a positive or negative offset), it is a length byte. If the amount of distinct values is low, it may be an enumerable. If the values are very distinct, it could be a data byte. This list is expected to be expanded as we encounter more data types and more protocols. Graph simplification methods may also be applied later.

Once the first step is over, one can read out the bytes from the root to each leaf to get each distinct message type. The algorithm at present can not be used to reverse the FSM itself, but it is part of the development plan.

3.2 A Prototype Implementation

The prototype implementation is developed in .NET, with a web-based front end written in ASP.NET MVC 5.



Currently available PCAPs

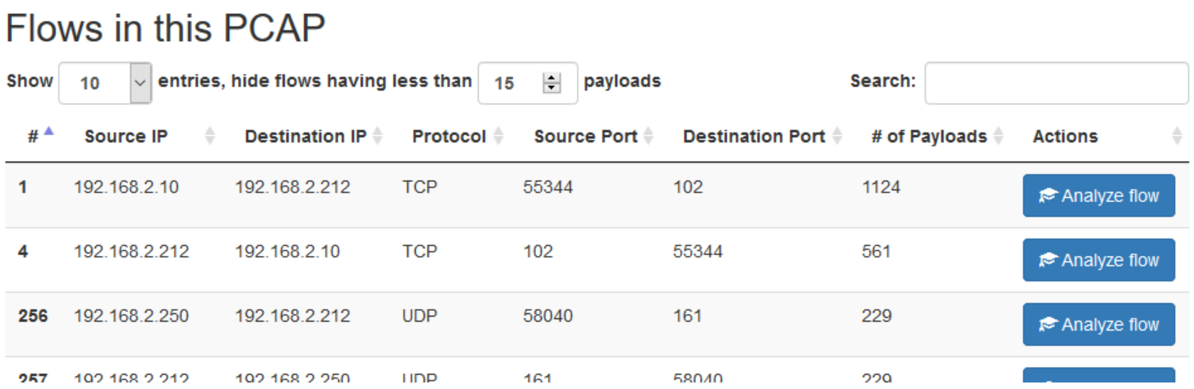
Show 100 entries Search:

#	File Name	# of Flows	Date Uploaded	Actions
1	S7_400_attack_3.pcap	538	2017.09.13. 16:03:12	View flows Delete
2	SMTP_Flow.pcap	16	2017.09.29. 01:23:49	View flows Delete

Showing 1 to 2 of 2 entries Previous 1 Next

Figure 1. PCAP selection

PCAP files from Wireshark or similar packet capture utilities may be uploaded on the dashboard (see Figure 1.), after which point, one may view the flows within.



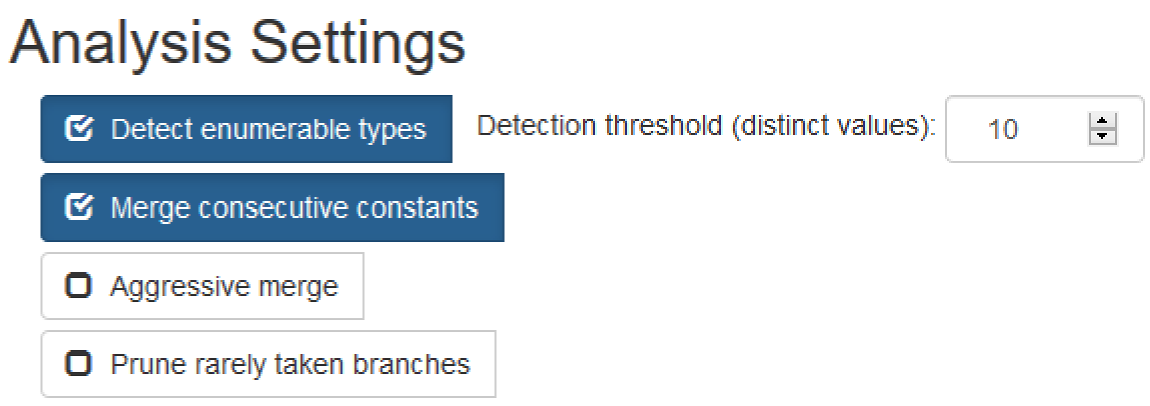
Flows in this PCAP

Show 10 entries, hide flows having less than 15 payloads Search:

#	Source IP	Destination IP	Protocol	Source Port	Destination Port	# of Payloads	Actions
1	192.168.2.10	192.168.2.212	TCP	55344	102	1124	Analyze flow
4	192.168.2.212	192.168.2.10	TCP	102	55344	561	Analyze flow
256	192.168.2.250	192.168.2.212	UDP	58040	161	229	Analyze flow
257	192.168.2.212	192.168.2.250	UDP	161	58040	229	

Figure 2. Flow selection

A flow can be defined as a distinct combination of the typical networking 5-tuple: source IP address, destination IP address, the transport layer protocol, the source port, and the destination port. In other words, a flow is a set of messages that were sent from the same sender to the same receiver in the same act of communication. These are listed on the second screen (see Figure 2.), where one should select the one they wish to analyze.



Analysis Settings

☑ Detect enumerable types   Detection threshold (distinct values): 10

☑ Merge consecutive constants

☐ Aggressive merge

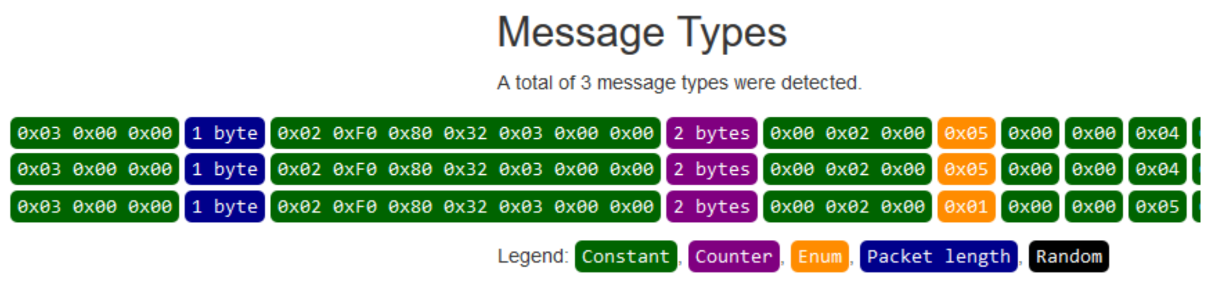
☐ Prune rarely taken branches

Figure 3. Customizing analysis settings

On the analysis screen (Figure 3.), it is possible to customize and fine-tune certain parameters of the algorithm. Currently, the analysis can be fine-tuned in four ways:

- The detection threshold for enumerable types is the number of distinct values under and equal to which a byte is considered an enumerable. Generally, values between 5 and 15 produce the best results, but this may depend on the actual use case.
- Enabling merging consecutive constants. This will convert consecutive one-byte constants into one single constant with a value that is equal to the values of the individual bytes, concatenated. For example, if there were three consecutive constants A, B, and C with the values *0x03*, *0x00*, and *0x00* respectively, they would be merged into one single node (of type constant) with the value *0x03 0x00 0x00*. This feature stops merging after reaching the first node that has two or more leaf nodes, in order to keep the values aligned. This makes parts of the graph easier to compare visually.
- Aggressive merging. With this option enabled, the previous merge operation does not stop processing when a node has two or more children.
- Pruning rarely taken branches. This feature deletes branches that would be created based on fewer messages than the threshold. Since analysis requires as many messages as possible on each branch in order to be accurate, branches with only a few messages will most likely not yield accurate results, and may be discarded.

Once the parameters are set, the analysis can be started by clicking *Start Analysis*.


Message Types

A total of 3 message types were detected.

0x03 0x00 0x00	1 byte	0x02 0xF0 0x80 0x32 0x03 0x00 0x00	2 bytes	0x00 0x02 0x00	0x05	0x00	0x00	0x04	
0x03 0x00 0x00	1 byte	0x02 0xF0 0x80 0x32 0x03 0x00 0x00	2 bytes	0x00 0x02 0x00	0x05	0x00	0x00	0x04	
0x03 0x00 0x00	1 byte	0x02 0xF0 0x80 0x32 0x03 0x00 0x00	2 bytes	0x00 0x02 0x00	0x01	0x00	0x00	0x05	

Legend: Constant, Counter, Enum, Packet length, Random

Figure 4. Analysis results - data types

When the analysis is complete, the results are displayed in two different forms. The first form (see Figure 4.) shows each recognized type in a different colour, along with values or lengths depending on what makes more sense for each type. The total number of message types is also indicated. At present, the current implementation can reliably detect and find constants, counters (of lengths 1, 2 or 4 bytes), length indicators (of length 1, 2, or 4 bytes), enumerable types, and data values.

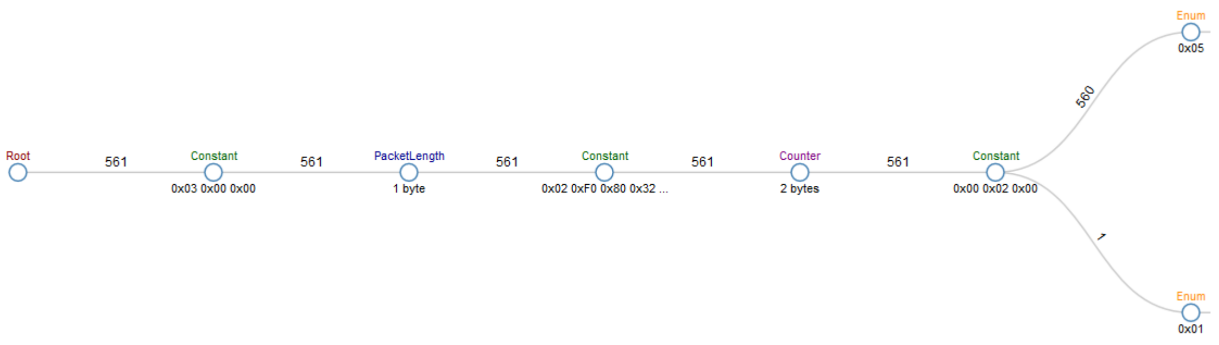


Figure 5. Analysis results - message graph

The second form is a graph (see Figure 5.) where the vertices are also coloured, and the data types are shown. In addition, the number of packets that were used to generate each branch is displayed. It is possible to click a vertex to collapse that branch (and hide everything else that follows said vertex), which may be useful when manually inspecting results. The graph is drawn using the d3.js library, which can automatically scale the drawing area and position the leaves such that they fill the available space.

4 Conclusion

As it can be seen from the preliminary results, our approach works. However, there is still a lot of research and development left. The algorithm needs to be tested on more protocols, then refined as necessary. We make some assumptions that may not always be true (for example, that everything is represented on whole bytes); the algorithm should be generalized to work even if these assumptions don't hold. We'll also need to find a way to represent the reverse-engineered protocol in a way that can be interpreted by other software, such as Wireshark. This area has plenty of unsolved issues and other challenges, and will be a good candidate for research for the next few years.

5 References

1) Marshall Beddoe. 2004. The protocol informatics project. Retrieved March 19, 2014 from http://www.4tphi. net/~awalters/PI/PI.html.
2) Weidong Cui, Jayanthkumar Kannan, and Helen J. Wang. 2007. Discoverer: Automatic protocol description generation from network traces. In USENIX Security Symposium.
3) Yipeng Wang, Xingjian Li, Jiao Meng, Yong Zhao, Zhibin Zhang, and Li Guo. 2011a. Biprominer: Automatic mining of binary protocol features. In 2011 12th International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT), 179–184.
4) Yipeng Wang, XiaoChun Yun, M. Zubair Shafiq, Liyan Wang, Alex X. Liu, Zhibin Zhang, Danfeng Yao, Yong Zheng Zhang, and Li Guo. 2012. A semantics aware approach to automated reverse engineering unknown protocols. In 2012 20th IEEE International Conference on Network Protocols (ICNP).
5) Joao Antunes, Nuno Neves, and Paulo Verissimo. 2011. Reverse engineering of protocols from network traces. In 2011 18th Working Conference on Reverse Engineering (WCRE), 169,178. DOI:10.1109/WCRE.2011.28
6) Jian-Zhen Luo, and Shun-Zheng Yu. 2013. Position-based automatic reverse engineering of network protocols. Journal of Network and Computer Applications 36, 3 (2013), 1070–1077.
7) Corrado Leita, Ken Mermoud, and Marc Dacier. 2005. ScriptGen: An automated script generation tool for HoneyD. In 21st Annual Computer Security Applications Conference (ACSAC'05), 200–214. DOI:10.1109/CSAC.2005.49.
8) Weidong Cui, Vern Paxson, Nicholas C. Weaver, and Randy H. Katz. 2006. Protocol-independent adaptive replay of application dialog. In Proceedings of the 13th Symposium on Network and Distributed System Security (NDSS'06).
9) Maxim Shevertalov and Spiros Mancoridis. 2007. A reverse engineering tool for extracting protocols of networked applications. In 14th Working Conference on Reverse Engineering (WCRE'07). 229–238. DOI:10.1109/WCRE.2007.6

10) Yipeng Wang, Zhibin Zhang, Danfeng Yao, Buyun Qu, and Li Guo. 2011b. Inferring protocol state machine from network traces: A probabilistic approach. Applied Cryptography and Network Security 2011.