Monograph of Program Analysis for Understanding and Reverse Engineering of Network Protocols

Sirshendu Ganguly
Worcester Polytechnic Institute
sganguly@wpi.edu

Justin Amevort
Worcester Polytechnic Institute
jamevor@wpi.edu

Cooper Bennet
Worcester Polytechnic Institute
cbennet@wpi.edu

ABSTRACT: Reverse engineering is an ever expanding area of research for network engineering. It is a process by which one can determine the inner functionality of a program while only having access to the outward facing functions of the process. This monograph analyzes the current state of reverse engineering in the domain of network engineering and evaluates several research papers on the topic. The included papers will be analyzed based on the ideas they introduce and how well they present the information for readers to follow. The aim of the monograph is to provide a summary of the current state of the technology available and to look at where the field is trending toward in the future. The monograph concludes that reverse engineering is becoming more automated but most applications still require a large amount of manual work as the systems for automatic reverse engineering are very general and need heavy modification for specific applications.

1. INTRODUCTION

This paper aims to propose a monograph by conducting a review and analysis of some of the popular reverse engineering techniques for the network protocols. The process of protocol reverse engineering is thought to be a sequence of three steps - *observation*, *preprocessing* and *inference* [8]. In the *observation* phase, the analyst places appropriate tools for collecting network traces from the environment that is being examined. The quality of the collected observations hugely depends upon the observation duration because a too short period of observation may lead to an incomplete message sequence thus hampering the entire protocol reverse engineering process. Since the observation phase may lead to obtaining some irrelevant data too, it becomes necessary to preprocess the data, in order to reconstruct the appropriate message. This is carried out in the *preprocessing* step. Thus, the preprocessing step must deal with identifying the execution trace of control structures that helps in identifying the encapsulation and iterative structure of the messages. The final step is *inference* which deals with identifying structures used in a message and builds the grammar based on which the messages are exchanged. This is a

crucial step because protocol specifications must be represented using appropriate models that resemble the original specification.

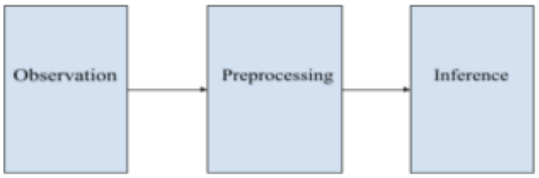

Figure 1: Protocol reverse engineering steps

For this monograph, we have reviewed and categorized several research papers on reverse engineering varying from manual reverse engineering to automatic reverse engineering. We have also studied some special cases dealing with encryption, and fuzzers. In our analysis, we examined the methodology described in the papers and checked whether their suggested implementations would work and if so, how effective they would actually be for their intended purpose.

The limitations of such analysis of reverse engineering techniques stem from fundamental aspects of the network security protocol. The wide availability of different reverse engineering tools each with its own implementation may produce slightly different interpretations of the source code. Thus one must be able to discern the information produced from the interpretation and spot imperfections. Moreover, the complexity of such analysis also increases while dealing with encrypted communication [1]. Some tools currently do address the problem of reverse engineering encrypted messages, but it is still a difficult task with a labor-intensive analysis process [2]. Also, the experiments to simulate real-world scenarios are limited because most studies examine a controlled protocol of one or a few messages of a typical routine.

We also encountered some challenges during preparing the monograph. The first is that a monograph focuses on a narrow subject area, which makes it hard to find a good number of high-quality sources to analyze. The ability to disseminate important information from code decompilation is critical to successful reverse engineering. Thus, we had to rely on readily available tools and software that is not meant to adequately simulate real systems. Another issue was that, depending on the characteristics of artifacts being analyzed, i.e., programming languages, pointers, and polymorphism, the interpretations vary. To address these problems, we grouped similar reverse engineering techniques together to ensure that we can compare them.

At first, we majorly reviewed a few manual reverse engineering techniques and pointed out how most of these techniques are cumbersome and error-prone. Thus moving forward to automatic reverse engineering techniques such as network-traffic-based and application-program-based. At last, we also evaluated some popular reverse engineering techniques that deal with encryption, etc.

In *Section 1*, we briefly described reverse engineering and spoke about its

limitations in general. *Section 2* presented a brief summary of different techniques used for reverse engineering varying from manual to automatic processes. *Section 3* elaborates on the description of our methodology and *Section 4* describes our evaluation of the different reverse engineering techniques. Finally, we summarize our results in *Section 5*.

2. LITERATURE REVIEW

In this section we discuss several research papers ranging from manual reverse engineering to automatic reverse engineering including techniques dealing with special cases such as encryption and middlebox management.

2.1 Manual Reverse Engineering

The most primitive method of reverse engineering is by manually evaluating a program. Research teams spent excessive hours analyzing a protocol in a labor intensive process. In each approach examined, the reverse engineering team took several days or weeks deciphering the protocol depending on its size.
Also, depending on the reverse engineering technique and protocol, the results may return obscure or difficult to understand [31]. Some enterprises elected this method based on specific business needs and circumstances. [30] Manual techniques were an original method for reverse engineering. In the past few years the industry has moved more towards automatic techniques because of its efficiency and time savings.

2.2 Automatic Reverse Engineering

An efficient alternative to manual reverse engineering is automatic reverse engineering. Automatic reverse engineering techniques for generating protocol specifications can be broadly classified into two categories based on the input it takes. The first category takes the network traffic as input, while for the second category the application program implementing the protocol serves as the input.

2.2.1 Network-traffic-based

In *network-traffic-based* protocol reverse engineering techniques, the network traces generated from recording the communication between a client and server is analyzed to look for occurrences of bytes that indicate a special meaning for the network protocol. Three popular instances of network-traffic-based reverse engineering techniques are described in [11]-[13]. In [11], the goal was to infer protocol idioms seen in message formats of many protocols by clustering messages with similar patterns from the network traces. While in [12], a real machine was put on the internet to act a honeypot, so that all the traffic from in or out of the machine could be recorded and analyzed to derive a state machine that represents the observed requests and replies. The state machine was simply later used to derive a script that can recognize incoming requests and provide suitable responses but the dependency handling offered was not upto the mark. Paper [13] is an improvement as it proposes an algorithm that infers dependencies in the context of protocol

messages without needing the knowledge about the semantics of the protocol.

2.2.1 Application-program-based

On the other hand, the *application-program-based* reverse engineering techniques like the ones described in [14]-[16] take the application program as input and use static or dynamic analysis to generate the set of inputs that this application program accepts. These approaches are motivated by the idea that the application program that itself encodes the protocol represents the format of. In [15], the approach leveraged dynamic taint analysis to monitor how an application program processes the messages that it receives. But, the approach described in [16] analyzed the buffers holding the messages used in communication as these buffers represent the inverse of the structure of the messages used for communication.

2.3 Special Cases

In this subsection, we discussed papers that dealt with special cases such as encryption and middle box management.

2.3.1 Encryption

Several of the papers reviewed worked to bypass encryption by analyzing the encrypted packets or programs. The various authors were attempting to find out the contents of encrypted messages without having the key. The common stated reason for doing this was for use against malware. Many forms of malware are advancing and utilizing encryption to make them harder to detect and harder for the intercepted messages to be read. By analyzing the packets and using various techniques to partially unencrypt the messages, the authors hoped to be able to combat these advanced malware threats. One method discussed as a way to use reverse engineering to decrypt text is to take tokens known to exist in the plain text version of the text and search for sets of characters in the encrypted file that could represent it with pattern analysis. One way to guarantee certain strings exist is to run the code given and see what the output or target is. Then the text has been revealed to contain that value in it and the reverse engineering techniques can be used to begin decryption.

2.3.2 Fuzzers

There are a few special cases that utilize reverse engineering techniques for specific application needs. A fuzzing tool is an automatic feature which detects program vulnerabilities by injecting randomized data into a program. In most cases a fuzzer can be used by an enterprise to help prove a program's correctness or highlight bugs in its source code. Various different types of fuzzers work depending on the information available to it.

2.3.3 Middlebox State Analyzer

Tools like state analyzers utilize reverse engineering techniques to reveal program specifications. StateAlyzer is a novel program used to provably and automatically identify all states that must be

migrated or cloned to ensure consistent middlebox outputs during dynamic redistribution [25]. StateAlyzer uses middlebox design intuition and static analysis techniques to deliver a powerful investigative tool for middleboxes.

3. METHODOLOGY

In this section, we describe how we analyzed and compare each of the described reverse engineering tools in *Section 2*. Since the authors of each of the mentioned papers had a different approach to protocol reverse engineering, it becomes a necessity to evaluate each one of them in a setting that is uniformed. The goal of the methods done will be to categorize the papers into relevant technical implementations and draw conclusions on their strengths and weaknesses.

To evaluate the current state of the industry we categorized the papers into broader classes based on the approach they used. The first two categories are *manual* and *automatic* reverse engineering. The third category was *special cases* that dealt with papers having special applications of reverse engineering in computer networking. We used these categories to identify relevant approaches by encompassing the tools' similarities to draw conclusions on their strengths and weaknesses.

4. EVALUATION

In this section we evaluated each of the papers belonging to the earlier mentioned categories.

4.1 Manual Reverse Engineering

Manual reverse engineering required a significant amount of manual labor by an engineering team when utilized. The team of 15 engineers spent approximately 150 hours each over a 3 week period reverse engineering BOS/X [30]. Figure 2 shows the cumulative time spent by each researching in meetings used to work on the project. The paper Discoverer shared two other examples of reverse engineering, both of which had extremely long development times and were not extendable to software updates.

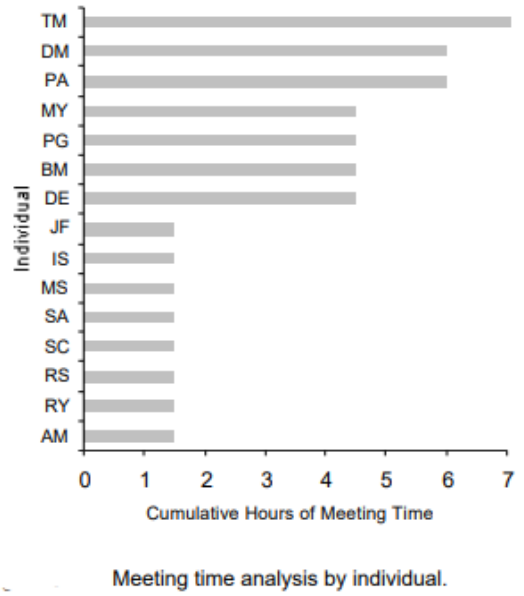

Figure 2: Time spent by researcher reverse engineering BOS/X

Compared to the other methods explored in this monograph, there were a limited amount of studies done on manual reverse engineering, likely because of the push to use more automatic methods. [30] provided valuable insight about the reengineering and analysis goals that

motivated the team's work. The paper also highlighted the realistic application of manual reverse engineering techniques for business use cases. However, the presentation of results were somewhat limited, specifically the graphs were general and did not present details about the analysis discovered by each researcher. They also did not have descriptive labels. For its figures and charts.

4.2 Automatic Reverse Engineering

4.2.1 Network-traffic-based reverse engineering

The interesting aspect of the approach described in [11] is that it operates in a protocol-independent fashion, by inferring protocol idioms seen in the application-level protocol message formats. The idea behind this approach is that the application-level protocols often share some common protocol idioms that correspond to the essential components in a protocol specification. The metrics used by the authors for evaluation of their approach are - correctness, conciseness and coverage which are sound. Correctness measures whether the inferred format corresponds to the true format. Conciseness measures the number of inferred formats that equals a single true format reflected in. And, coverage measures how many messages are covered by the inferred formats. However, their results showed a significant difference between the message and format coverage which they blamed on the heavy-tail distribution of message formats.

Moving forward to the approach described in [12], no assumptions are made about the nature of the protocol so it can be applied to a wide range of different protocols as no prior knowledge of the protocol's behavior is required. At first, a real machine was put out on the internet so that all the traffic entering or exiting the machine could be recorded. The traffic was later analyzed to derive a state machine that represents the observed requests and replies which was used to derive a script that can recognize the incoming requests and thus provide suitable responses. However, this approach suffered as the dependency handling in the context of protocol messages was not efficient and increased the overall complexity of the process.

The paper [13] provides a commendable improvement over the earlier discussed approaches as it is capable of inferring dependencies in the context of protocol messages without needing knowledge about the semantics of the protocol. As opposed to the earlier approaches, the techniques used by the authors deliberately refused to take advantage of any heuristic to recognize important fields in the arguments received from the clients or sent by the servers. Instead, they used several instances of the same type of messages, to automatically retrieve the fields which have some importance from a semantic point of view and are important to let the conversation between the client and server continue.

Unfortunately, most of these network-traffic-based reverse engineering

techniques fail and had limited precision because of the lack of information in the network traces which often leads to classifying messages of the same type as different. Also, these techniques can easily be hampered by obfuscation or encryption.

4.2.1 Application-program-based

The authors of the paper [15] summarized many papers relating to protocol reverse engineering methods and their history. This explanation is a good background that shows they understand the topic and helps readers to understand the material. The experiments that were conducted sampled from widely used protocols that utilized a mix of text and binary protocols. For each protocol, multiple test cases were written to check their ability to automatically detect protocol specifications. From their evaluation, it appears that they were able to detect a large amount of the protocols that they wrote test cases for. However, the protocols that they chose to test with contain much more information than the few parts they tested on. The overall system seems like it would work well for general use provided that it does not have large runtime or memory usage, neither of which is supported in their approach.

On the other hand, the paper [16] provided extensive background information about approaches to automatically reverse engineer and the difficulty in rewriting messages sent and received through a network. The experiment the authors conducted was thoughtful to utilize previous discoveries and substantially limit the rewriting process in message transfers. And their results show that in various test cases, their approach outperformed another popular reverse engineering tool called Wireshark.

However, these approaches have limitations too as it is not possible to generate the sets of all inputs that the application program operates on using dynamic analysis. Moreover, these techniques suffer from scalability issues too.

4.3 Special Cases

4.3.1 Encryption

[19] summarizes many papers relating to protocol reverse engineering methods and its history. This explanation is a good background that shows they understand the topic and helps readers to understand the material. The experiments that were conducted sampled from widely used protocols that utilized a mix of text and binary protocols. From their evaluation it appears that they were able to detect a large amount of the protocols that they wrote test cases for, however the protocols that they chose to test with contain much more information than the few parts they tested on. The system seems like it would work well for general use provided that it does not have large runtime or memory usage, neither of which is provided in their paper.

App Category	Stats	Params	Pairs	Perm	Comb	TotalTCs	PairsCov	PermCov	CombCov	TotalCov	Uncov	Dur
Official	Total	227	243	169	68	480	55	8	6	69	11	25.27
	Mean	7.57	8.10	5.63	2.27	16.00	1.83	0.27	0.20	2.30	0.37	0.84
	Median	7.00	4.00	0.00	0.00	5.00	1.50	0.00	0.00	2.00	0.00	0.56
	Std Dev	3.81	10.76	11.78	12.04	21.44	1.46	0.83	0.55	1.51	0.67	0.78
Third Party	Total	67	41	0	0	41	43	0	0	43	0	2.76
- Benign	Mean	6.70	4.10	0.00	0.00	4.10	4.30	0.00	0.00	4.30	0.00	0.28
	Median	5.50	4.00	0.00	0.00	4.00	2.50	0.00	0.00	2.50	0.00	0.12
	Std Dev	4.42	2.96	0.00	0.00	2.96	4.55	0.00	0.00	4.55	0.00	0.38
Third Party	Total	157	149	234	12	395	82	15	1	98	13	24.06
- Anomalous	Mean	7.85	7.45	11.70	0.60	19.75	4.10	0.75	0.05	4.90	0.65	1.20
	Median	7.50	6.50	1.00	0.00	10.50	3.00	0.00	0.00	4.00	1.00	2.00
	Std Dev	4.06	3.15	20.50	1.96	22.69	3.26	1.65	0.22	2.85	0.67	0.91
Overall	**Total**	451	**433**	**403**	**80**	**916**	**180**	**23**	**7**	**210**	**24**	52.09
	Mean	7.52	7.22	6.72	1.33	15.27	3.00	0.38	0.12	3.50	0.40	0.87
	Median	7.00	6.00	0.00	0.00	8.50	3.00	0.00	0.00	3.00	0.00	0.79
	Std Dev	3.95	7.97	14.84	8.57	20.53	3.00	1.14	0.42	2.90	0.64	0.83

Params refers to number of parameters; **Pairs** refers to number of pairwise test cases; **Perms** refers to number of permutation test cases; **Comb** refers to number of all-combinations test cases; **TotalTCs** refers to total number of test cases; **PairsCov** refers to number of sinks covered by pairwise test cases; **PermCov** refers to number of sinks covered by permutation test cases; **Comb Cov** refers to number of sinks covered by all-combinations test cases; **TotalCov** refers to number of sinks covered by all the test cases; **Uncov** refers to number of sinks that are not covered; **Dur** refers to the test execution duration in hour.

Table 2: Experimental Results of Smartfuzz on testing 60 Smartapps

In [28] the authors do a great job of providing adequate background on the topic to readers. They provide a summary of the target malware as well as the different approaches to the problem they attempt to solve. Their system was able to very effectively decrypt data using a verification identification method on AES primitives, detecting 94.6% of the primitives. The system they propose has several limitations such as being detectable by malware, and thus avoided, as well as requiring the malicious code to execute in order for it to be evaluated. Their system was designed to primarily target C and C++ code so it has very limited effectiveness against other programming languages.

4.3.2 Fuzzers

In this category, the papers observed presented reverse engineering techniques for specific network protocols. As Iot devices continue to become popular and grow, understanding different tools efficiency and usefulness was a goal for the research team. Fuzzers present themselves as powerful tools to identify program vulnerabilities.

Table 2 displays the experimental results of SmartFuzz. The table presents how SmartFuzz efficiently and effectively produces test cases. SmartFuzz performed well compared to other Fizzers like *RandFuzz,* that did not have as much sink coverage. The growth of the reverse engineering industry has evolved to include robust tools for use. In each paper examined, the fuzzers showed effectiveness for the majority of use cases. Fuzzers are a powerful and intuitive tool in the market currently that can help make systems more secure. Their ease of use and simple implementation make them ideal for testing various types of systems.

4.3.2 Middlebox State Analyzers

StateAlyzer also had positive results from its evaluation. The program effectively identifies middlebox states and works on complex programs. The importance of state analyzers like StateAlyzer is vital because it is difficult and requires manual labor to identify state locations for migration or cloning. StateAlyzer ran comparatively well to author reverse engineering techniques including run time analysis, symbolic

execution, and manual code inspection. In each case Statelyzer ran quicker or more efficiently than other implementations [25].

5. CONCLUSION

We have prepared a monograph on the topic of program analysis techniques used to understand and reverse engineer network protocols. We first started with the manual reverse engineering techniques which soon were replaced with automatic implementations. We also briefly discussed the difference between network-traffic based automatic protocol reverse engineering techniques and application-level-based automatic reverse engineering techniques. [15] Leveraged dynamic taint analysis to monitor how messages are processed in order to reverse engineer the underlying protocol. Another approach we discussed as described in [16] processed messages but did not rely on dynamic tainting, instead understood protocol specification from the structure of the buffers used to store these messages. Another interesting paper [17], discussed the challenges of reverse engineering encrypted messages thus they proposed to identify runtime buffers in order to observe how the application program parses and processes a message. [22] discusses dynamic binary analysis as a revolutionary weapon for the extraction of the protocol message format. We also reviewed a technique proposed in [18] which uses clustering on network traces to reverse engineer protocols. The paper [25] has a shift of interest as its goal was to automatically identify all the states that must be migrated or cloned to keep the middlebox output consistent. The paper [26] and [28] was revolutionary because it was capable of automatically discovering the encryption algorithm and key, thus was capable of generating a protocol specification for the command and control used by the bonnet. Paper [27] is another solid example of how suspicious activity can be detected by passively monitoring the network traffic.

REFERENCES

[1] Wang, Zhi & Jiang, Xuxian & Cui, Weidong & Wang, Xinyuan & Grace, Mike. (2009). ReFormat: Automatic Reverse Engineering of Encrypted Messages. 5789. 200- 215. 10.1007/978-3-642-04444-1_13.

[2] Comparetti, Paolo & Wondracek, Gilbert & Krügel, Christopher & Kirda, Engin. (2009). Prospex: Protocol Specification Extraction. Proceedings - IEEE Symposium on Security and Privacy. 110-125. 10.1109/SP.2009.14.

[3] Caballero, Juan & Poosankam, Pongsin & Kreibich, Christian & Song, Dawn. (2009). Dispatcher: Enabling active botnet infiltration using automatic protocol reverse-engineering.621-634.10.1145/1653662.1653737.

[4] Wondracek, Gilbert & Comparetti, Paolo & Krügel, Christopher & Kirda, Engin. (2008). Automatic Network Protocol Analysis.

[5] Chiang, Ken & Lloyd, Levi. (2007). A case study of the rustock rootkit and spam bot. 10-10.

[6] Cui, Weidong & Paxson, Vern & Weaver, Nicholas & Katz, Randy. (2006). Protocol-Independent Adaptive Replay of Application Dialog..

[7] Lin, Zhiqiang & Jiang, Xuxian & Xu, Dongyan & Zhang, Xiangyu. (2008). Automatic Protocol Format Reverse Engineering through Context-Aware Monitored Execution.

[8] Duchêne, Julien & Guernic, Colas & Alata, Eric & Nicomette, Vincent & Kaaniche, Mohamed. (2018). State of the art of network protocol reverse engineering tools. Journal of Computer Virology and Hacking Techniques. 14. 10.1007/s11416-016-0289-8.

[9] X. Li and L. Chen, "A Survey on Methods of Automatic Protocol Reverse Engineering," in 2013 Ninth International Conference on Computational Intelligence and Security, Sanya, Hainan China, 2011 pp. 685-689. doi: 10.1109/CIS.2011.156

[10] Duchêne, Julien & Guernic, Colas & Alata, Eric &Nicomette, Vincent & Kaaniche, Mohamed. (2018). State of the art of network protocol reverse engineering tools. Journal of Computer Virology and Hacking Techniques. 14. 10.1007/s11416-016-0289-8.

[11] Cui, Weidong, Jayanthkumar Kannan and Helen J. Wang. “Discoverer: Automatic Protocol Reverse Engineering from Network Traces.” USENIX Security Symposium (2007).

[12] Leita, Corrado & Mermoud, K. & Dacier, Marc. (2006). ScriptGen: an automated script generation tool for Honeyd. 12 pp.-. 10.1109/CSAC.2005.49.

[13] Leita, Corrado et al. “Automatic Handling of Protocol Dependencies and Reaction to 0-Day Attacks with ScriptGen Based Honeypots.” RAID (2006).

[14] Newsome, James & Brumley, David & Franklin, Jason & Song, Dawn. (2006). Replayer: Automatic protocol replay by binary analysis. 311-321. 10.1145/1180405.1180444.

[15] Wondracek, Gilbert & Comparetti, Paolo & Krügel,Christopher & Kirda, Engin. (2008). Automatic Network Protocol Analysis.

[16] Caballero, Juan & Poosankam, Pongsin & Kreibich, Christian & Song, Dawn. (2009). Dispatcher: Enabling active botnet infiltration using automatic protocol reverse-engineering. 621-634. 10.1145/1653662.1653737.

[17] Zhi Wang, Xuxian Jiang, Weidong Cui, Xinyuan Wang and Mike GraceReformat: Automatic Reverse Engineering of Encrypted Messages (2008)

[18] Caballero, J., Song, D.: Polyglot: Automatic Extraction of Protocol Format using Dynamic Binary Analysis. In: Proceedings of the 14th ACM Conference on Computer and and Communications Security (CCS’07). (2007)

[19] Wondracek, G., Comparetti, P.M., Kruegel, C., Kirda, E.: Automatic Network Protocol Analysis. Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS’08) (February 2008)

[20] Lin, Z., Jiang, X., Xu, D., Zhang, X.: Automatic Protocol Format Reverse Engineering Through Context- Aware Monitored Execution. Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS’08) (February 2008)

[21] Cui, W., Peinado, M., Chen, K., Wang, H.J., IrunBriz, L.: Tupni: Automatic Reverse Engineering of Input Formats. Proceedings of the 15th ACM Conferences on Computer and Communication Security (CCS’08) (October 2008)

[22] Lin, Z., Jiang, X., Xu, D., Zhang, X.: Automatic Protocol Format Reverse Engineering Through Context- Aware Monitored Execution. Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS’08) (February 2008)

[23] Cui, W., Peinado, M., Chen, K., Wang, H.J., IrunBriz, L.: Tupni: Automatic Reverse Engineering of Input Formats. Proceedings of the 15th ACM Conferences on Computer and Communication Security (CCS’08) (October 2008)

[24] Comparetti, P.M., Wondracek, G., Kruegel, C., Kirda, E.: Prospex: Protocol Specification Extraction. In: Proceedings of 2009 IEEE Symposium on Security and Privacy, Oakland, CA (May 2009)

[25] Khalid, J., Gember-Jacobson, A., Michael, R., Abhashkumar, A., & Akella, A. (2016).

StateAlyzr: Deep Diving into Middlebox State to Enable Distributed Processing.
[26] L. De Carli, R. Torres, G. Modelo-Howard, A. Tongaonkar and S. Jha, "Botnet protocol inference in the presence of encrypted traffic," IEEE INFOCOM 2017 -IEEE Conference on Computer Communications, 2017, pp. 1-9, doi: 10.1109/INFOCOM.2017.8057064.
[27] Paxson, V. (1998). Bro: a system for detecting network intruders in real-time. Comput. Networks, 31, 2435-2463.
[28] Gröbert F., Willems C., Holz T. (2011) Automated Identification of Cryptographic Primitives in Binary Programs. In: Sommer R., Balzarotti D., Maier G. (eds) Recent Advances in Intrusion Detection. RAID 2011.
[29] Serge Gorbunov and Arnold Rosenbloom. 2010. AutoFuzz: (2010) Automated network protocol fuzzing framework. International Journal of Computer Science and Network Security 10, 8, 239--245
[30] Swafford Dave, Elman Diana, Aiken Peter, and Merhout Jeff. 2000. Experiences reverse engineering manually 3-7
[31] Carlsbad Assaf, Moving From Manual Reverse Engineering of UEFI Modules To Dynamic Emulation of UEFI Firmware, 2020
[32] Khin Shar Lwin, Binh Duong Ta Nguyen Jiang Lingxiao, Lo David SmartFuzz: An Automated Smart Fuzzing Approach for Testing SmartThings Apps, 2020 365-367