PRETT: Protocol Reverse Engineering Using Binary Tokens and Network Traces

Choongin Lee, Jeonghan Bae, and Heejo Lee(✉)

Korea University, Seoul, Republic of Korea
{choonginlee,permion,heejo}@korea.ac.kr

Abstract. Protocol reverse engineering is the process of extracting application-level protocol specifications. The specifications are a useful source of knowledge about network protocols and can be used for various purposes. Despite the successful results of prior works, their methods primarily result in the inference of a limited number of message types. We herein propose a novel approach that infers a minimized state machine while having a rich amount of information. The combined input of tokens extracted from the network protocol binary executables and network traces enables the inference of new message types and protocol behaviors which had not been found in previous works. In addition, we propose a state minimization algorithm that can be applied to real-time black-box inference. The experimental results show that our approach can infer the largest number of message types for file-transfer protocol (FTP) and simple mail-transfer protocol (SMTP) compared to eight prior arts. Moreover, we found unexpected behaviors in two protocol implementations using the inferred state machines.

Keywords: Protocol reverse engineering
State machine reconstruction · Automatic protocol analysis

1 Introduction

Protocol reverse engineering is the process of extracting application-level protocol specifications. These specifications provide researchers with knowledge about network protocols for multiple uses. First, specific information about protocols can help network protocol fuzzers launch a fuzzing process [12,18,22] to identify potential vulnerabilities. Detailed information about protocol messages and its state machine enables fuzzers to detect possible crashes in an implementation of a protocol. Simulating and replaying protocols in various environments [9,14,20] require the knowledge of specific protocol behaviors. Protocol specification is also necessary for protocol analyzers. Protocol analyzers are implemented using protocol parsers [3,16], which require detailed information about network messages. Finally, the knowledge of a protocol helps intrusion detection systems (IDSs) [11,23] to analyze malicious network patterns. Modern IDSs use deep packet

L. J. Janczewski and M. Kutyłowski (Eds.): SEC 2018, IFIP AICT 529, pp. 141–155, 2018.
https://doi.org/10.1007/978-3-319-99828-2_11

inspection (DPI) techniques that require dissecting protocol message formats and protocol state information [21].

From a security perspective, it is effective to prevent potential attacks by discovering the vulnerabilities that a protocol implementation can have using models in software testing. Testers can traverse all states as much as possible, using messages that can be used in the protocol, and test desired and unexpected inputs in each state. To this end, the goal of this research is to reconstruct a minimized state machine that contains rich information of a target protocol, similar to most other protocol reverse engineering studies.

Prior works have been conducted using two methods: static analytical method and dynamic analytical method. The static analytical method typically uses protocol network traces as the input for message syntax extraction [8,17,28] and state machine reconstruction [2,24,26]. In contrast, the dynamic analytical method uses the application data of the target protocol and human-provided messages as the input for message syntax extraction [4,5,10,25] and state machine reconstruction [6,7,15,27,29].

However, even with their successful inference, both static and dynamic methods still infer a limited number of protocol message types. This is because most of the existing methods [2,7,24,26,27,29] rely on how much information is contained in the input traces (network traces or application session traces) for the inference of message types. Most of the input traces are composed of messages and their frequently used sequences. Therefore, it is difficult to infer other types of messages that are not included in the input sources. Other types of approaches by Cho [6] and LaRoche [15] do not use traces as the input; instead, they use inputs that require human knowledge such as input alphabets and command sets. In this case, one cannot feed every possible input but can only input messages that are frequently used to the system, which results in a limited number of messages inferred.

We herein propose a novel approach that infers a minimized state machine with a rich amount of information, especially various types of protocol messages. The key idea is to use the combination of network traces with binary tokens as the input to find more message types than solely using network traces, application data, or human-provided messages as the input. Moreover, we apply two principles to infer a minimized state machine in real time while having the rich information. The comparative result with prior works shows that we could infer the largest number of message types for FTP and SMTP from eight approaches.

The contributions of this study are as follows:

- A protocol reverse engineering technique using network traces and message tokens (PRETT), which is a novel technique that infers a protocol state machine composed of a large quantity of text-based protocol message types is proposed.
- A state minimization algorithm with two principles motivated by the conventional automata theory is proposed to infer a minimized state machine while eliminating redundant states. The state minimization algorithm can be applied in a real-time black-box environment.

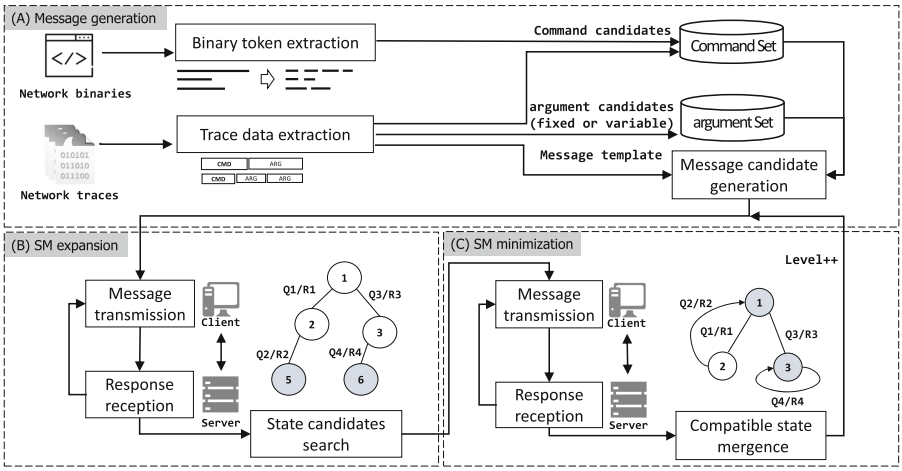


Fig. 1. Overall architecture of PRETT

– Separate state machines could be inferred using our approach, depending on their implementations of the same protocol. Moreover, PRETT observed unexpected behaviors in two protocol implementations, `Postfix` and `ProFTPD`.

2 Overall Description

Problem Scope. Protocol reverse engineering consists of two primary processes: message format inference and state machine inference. We focus on the latter to overcome the limitation of prior works. We referred to a technique presented by a recent study [14] to infer the message formats for each protocol. We also focus on the inference of text-based protocols. Text-based protocols are widely used because they are easy to test and debug (transparency), simple for tool writing (interoperability), and allow the easy expansion of existing command arguments or the addition of optional arguments (extensibility).

Terminology. Most messages of the text-based protocols consist of message commands and arguments. A *command* of messages is a major role in directing the processing of protocols. Message commands may be accompanied by their corresponding *arguments*. One or more arguments provide each protocol message with informative data to be processed by its counterpart. The *protocol state machine* represents the sequence in which protocol messages are communicated. The *state* in a protocol state machine is the status in which specific messages can be transmitted or received. Only specific messages in a state cause a *transition* from a state to another. Each *message type* is a unique message that causes different behavior in each state. For the comparative evaluation, we postulate that the number of message types of text-based protocols depends on the number of different commands.

Table 1. String refining process for obtaining protocol message tokens

Type	(1) Raw strings	(2) Tokenized strings	(3) Result tokens
Simple words	user, proc	user, proc	user, proc
File path	/lib64/ld-linux-x86-64.so	lib64, ld, linux, x86, 64, so	lib, ld, linux, so
Informative sentence	error: unexpected filename: %s	error, unexpected, filename, s	error, unexpected, filename
Function name	__stack_chk_fail, _exit	stack, chk, fail, exit	stack, chk, fail, exit
Variables	__progname	progname	progname
ELF library section	init_array, ġnu_debuglink	init, array, gnu, debuglink	init, array, gnu, debuglink
URL	http://www.gnu.org/licenses/gpl.html	http, www, gnu, org, licenses, gpl, html	http, www, gnu, org, licenses, gpl, html

Overall Architecture. The overall PRETT process is depicted in Fig. 1. PRETT starts by creating message candidates using protocol message tokens and network traces. Subsequently, it performs state machine expansion and state machine minimization for each level based on a tree model. In the state machine expansion step, every message candidate is transmitted to the server where the protocol is implemented. As different responses are observed, each next-state candidate is found while recording the pair of messages transmitted and received. Subsequently, the state machine minimization step merges the states by judging the compatibility between each next-state candidate and other states.

3 State Machine Inference

3.1 Message Generation

Our system requires a set of protocol message candidates that can be processed in an implementation (i.e., server program). Some of them might trigger state transitions to the next state and others to the state itself depending on the protocol implementation. Assuming that the target is a text-based protocol, three components are required to formulate the message candidates: commands, arguments, and templates. However, as one can not fully know all of them in the target protocol, their candidates need to be prepared from external sources. One source is the binary (program) implementing the target protocol, which provides message tokens that might be valid commands. The other source is the network trace of the target protocol, which contains the message templates, commands, and arguments of the message.

Binary Token Extraction. To obtain some hints of the commands from the binary executable, it first extracts raw strings from them. Subsequently, the strings are refined because of unnecessary characters. A description of the step-by-step string refining process is depicted in Table 1.

– Initially, raw strings are extracted from binaries. The example strings have various kinds of formats and are shown in the second column.

– Subsequently, the raw strings are split using special characters such as a space and a slash (/) as delimiters. The examples are shown in the third column. However, some of the tokens still cannot be used to generate protocol message candidates because they contain digits (e.g., *lib64, x86, 64*) or are only a single character (e.g., *s*).
– Finally, by the assumption that the commands of text-based protocols consist of multiple alphabetical characters, tokens that are composed of two or more alphabetic characters remain as command candidates. The results of the tokens are shown in the fourth column.

Trace Data Extraction. Each message transmitted from the client to the server is split with space separators into one or more keywords, and every first keyword is treated as a command and the following keywords as arguments considering the characteristics of text-based protocols [19]. The commands and arguments are acquired for the generation of message candidates. For the protocol template, we refer to the template acquisition method introduced in PRISMA [14].

Message Candidate Generation. Given the message templates of a protocol and both its command and argument candidates, message candidates are generated by combining a command candidate and one or more argument candidates depending on the templates. A template may have only one command field or one or more argument fields along with the command field.

An example of generating an SMTP message candidate is shown in Fig. 2. The example shows a template with one command field and two variable argument fields. According to the general property of text-based protocol messages, each field in a template is separated by a space and the message ends with a carriage return (CR) and line feed (LF). Command candidates obtained from either binaries or network traces are assigned to the command field, whereas argument candidates obtained from network traces are assigned to the variable argument fields. It is noteworthy that `STARTTLS` and `CONNECT`, which are marked bold, are difficult to obtain from network traces or human-provided inputs as in previous studies, but are readily available from binary tokens.

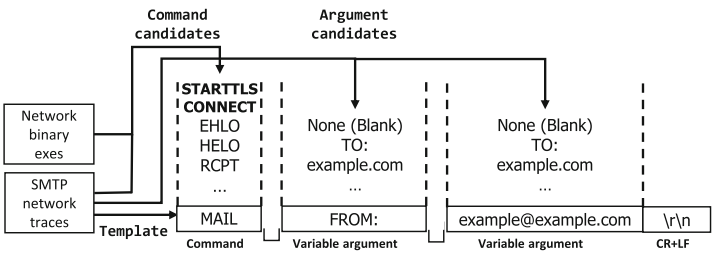


Fig. 2. Generation of SMTP message candidates using a template, command candidates, and argument candidates

3.2 State Machine Expansion

PRETT reconstructs a state machine based on a tree structure, in which a root node is considered an initial state. Given the message candidates, PRETT expands a state machine to have as many states as possible. Hence, every possible message candidate is transmitted to a server implementation and all responses are checked on every node at each tree level. Each response is compared with the responses previously observed in the same level, and a new state candidate is added to a set of state candidates if a distinct response is observed.

Two-Step Message Transmission. PRETT utilizes a two-step approach as follows to exclude redundant message candidates among possible message candidates.

- Initially, PRETT transmits command-only messages consisting of each command candidate first and it remembers commands that trigger each distinct response. The commands can be considered as valid commands that can be understood distinctly by the counterpart.
- Subsequently, composite messages that combine one or more arguments with the valid commands from the previous step are transmitted and PRETT remembers those that trigger each distinct response.

Resettability. To transmit a message in a specific state, a message sequence traversing from the initial state to the corresponding state is transmitted in advance. After transmitting each message in a specific state, a reset message is followed to set the server back to the initial state. The reset message is obtained by human observation on the trace. In a text-based protocol, a message such as `QUIT` is used to return to the initial state, which humans can understand.

3.3 State Machine Minimization

Given a set of state candidates in each tree level, their number needs to be reduced while only valid states are maintained. It is important to reduce the number of redundant states because the state machine expansion is performed again in each candidate later, in which the time complexity is proportional to the number of state candidates.

Principles of Compatibility Test. Among the state candidates, the redundant state candidates can be reduced using the compatibility test between each state candidate and the *valid* states. Such valid states refer to the initial state or the states remaining after the state-compatibility test. For the test, PRETT uses two principles as follows. The principles are motivated by a conventional automata theory [13].

Principle 1 (compatibility between states)**.** *If all message pairs from state A match all those from state B, then state A and B are compatible.*

Principle 2 (incompatibility between states)**.** *If there are distinct response messages to the same request message from state A and B, then state A and B are incompatible.*

Algorithm 1. State machine minimization

Input: S_n : a set of state candidates in the level n
Output: S_L : a set of valid nodes for current level

```
1  p ← 0                                   // initialize parent node
2  valid ← FALSE                           // flag for state validity
3  foreach s ∈ S_n do
4      valid ← FALSE
5      p ← get_parent(s)
6      M_s ← replay_messages_of(p)
7      if M_p = M_s then
8          merge(p, s) and S_L ← S_L − {s}
9      else
10         valid ← TRUE
11         foreach s' ∈ S_L − {s} do
12             p' ← get_parent(s')
13             if p' = p then
14                 M_s' ← retrieve_message_pairs()
15                 if M_s' = M_s then
16                     merge(s', s), S_L ← S_L − {s} and valid ← FALSE
17                     break
18         if valid = TRUE then
19             foreach s' ∈ S − {s, p} do
20                 p' ← get_parent(s')
21                 if p' ≠ p then
22                     M_s' ← replay_messages_of(s)
23                     if M_s' = M_s then
24                         merge(s', s), S_L ← S_L − {s} and valid ← FALSE
25                         break
26 return S_L
```

Three-Step Minimization. For the convenience of explanation, we assign several names to each node in the tree according to its characteristics. The state candidates at each level are termed as *subnodes*. A *parent node* of a subnode is a valid node from which a transition occurs to the subnode directly. A *sibling node* of a subnode is a valid node that has the same parent node. Finally, a *relative node* is all the valid nodes in a tree except the parent and sibling nodes. The reason for classifying each node in the tree is to prioritize the compatibility test. By empirical observations, many cases exist where each state candidate is compatible with the parent, sibling, and relative node, in that order.

Algorithm 1 describes the formal process of state machine minimization at each tree level. The process is divided into three steps: compatibility test with parent, sibling, and relative node.

- *Compatibility test with parent node.* For each subnode and its parent node, all transmitted messages from the parent node to its subnodes are replayed in the subnode (line 6). It then checks the pair of transmitted and received messages. If the message pairs on the parent node and those on the subnode are the same based on principle 1 (line 7), then the subnode is merged with its parent node.
- *Compatibility test with sibling node.* If the message pairs on a sibling node that are retrieved from the previous record (line 14) and those on the subnode are the same (line 15), then the subnode is merged with the sibling node (line 16).
- *Compatibility test with relative node.* In each relative node new message pairs are obtained by replaying all the messages transmitted in the message pairs of each subnode (line 22). If the message pairs on a relative node and those on the subnode are the same (line 23), then the subnode is merged with the relative node (line 24). Otherwise, the subnode remains as a valid node in the level.

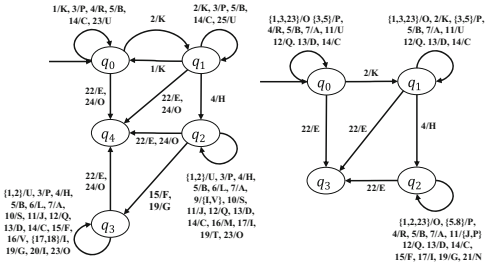
End Condition. After all subnodes are tested for the compatibility with the other valid nodes through the three steps, PRETT produces a set of valid states found in the level. Each valid state is used as a base state for the state expansion of the next level. In particular, when no valid state is present as a result of state minimization in the level, no state expansion and minimization process needs to be performed and a minimized state machine that is composed of only valid states is returned.

4 Evaluation

To ensure that PRETT can infer the state machines for real application-level protocols effectively, we applied our state machine inference mechanism to some applications in several protocols. We chose FTP, SMTP, and HTTP as target protocols, whose applications are widely deployed. We chose implementations to reverse engineer the target protocols, which are `vsftpd` (version 3.0.3) and `ProFTPD` (version 1.3.5a) for FTP, `Postfix` (version 3.1.0) for SMTP, and `Apache` (version 2.4.18) for HTTP.

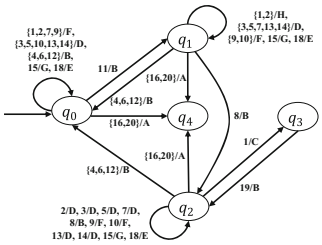
Table 2. Summary of input sources for the experiment

Protocol	Binary	Trace	Token	Argument
FTP	4 (ftp, netkit-ftp, pftp, wget)	118 (web crawling)	2017 (binary), 19 (trace)	27 (trace)
SMTP	2 (mailutils, sendmail)	51 (web crawling)	3062 (binary), 9 (trace)	25 (trace)
HTTP	3 (openssl, webbrowser-app, wget)	232 (web crawling)	5135 (binary), 2 (trace)	228 (trace)



Request message table				Response message table			
ID	Message	ID	Message	ID	Message	ID	Message
1	USER	13	SYST (argument)	A	200	M	425
2	USER Anonymous	14	HELP (argument)	B	211	N	450
3	OPTS (argument)	15	PASV (argument)	C	214	O	500
4	PASS (argument)	16	RETR (argument)	D	215	P	501
5	FEAT (argument)	17	XCUP	E	221	Q	502
6	REST (argument)	18	XCUP [argument]	F	227	R	503
7	NOOP (argument)	19	EPSV	G	229	S	504
8	PORT (argument)	20	EPSV [argument]	H	230	T	522
9	CWD (argument)	21	STOU (argument)	I	250	U	530
10	MODE (argument)	22	QUIT	J	257	V	550
11	PWD (argument)	23	XXX (Random commands)	K	331		
12	ACCT (argument)	24	HTTP commands	L	350		

Fig. 3. Inferred FTP state machine



Request message table				Response message table	
ID	Message	ID	Message	ID	Message
1	DATA	11	MAIL FROM:<address>	A	221
2	DATA [argument]	12	RSET	B	250
3	EHLO	13	RSET [argument]	C	354
4	EHLO [argument]	14	VRFY	D	501
5	HELO	15	VRFY [argument]	E	502
6	HELO [argument]	16	QUIT	F	503
7	RCPT (argument)	17	STARTTLS	G	550
8	RCPT TO:<address>	18	XXX (Random commands)	H	554
9	AUTH (argument)	19	[Mail content]		
10	MAIL (argument)	20	HTTP commands		

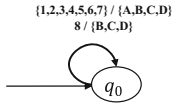
Fig. 4. Inferred SMTP state machine

Test Inputs. Table 2 shows the number of each type of inputs for the protocol state machine inference used in the experiment. The input binaries are obtained from two sources: Linux built-in distribution (e.g., Ubuntu desktop) and an external open-source repository. It is noteworthy that the inference is possible with client binaries implementing the protocol even without the server binaries of the protocol. The traces are obtained from web crawling by searching for pcap files that capture only each protocol. The primary source for the pcap files is the website *pcapr* [1]. We used a subset of the arguments such as words that consist of alphabetical characters only, directory paths, numbers, and email addresses to reduce the running time during the experiment.

4.1 Inferred State Machine

[1]**File-Transfer Protocol.** In using the FTP, users may authenticate themselves with a username and password, but can connect in an anonymous mode if the server is configured to allow it. The inferred state machines for two FTP servers inferred by PRETT are described in Fig. 3(a) `vsftpd` and (b) `ProFTPD`. The state machine shows that PRETT can infer the complex communication of the

[1] For all the figures of the inferred state machines, the labels on each edge denote a pair of the transmitted-received message. The table on the right shows the inferred message types for request and response. The request messages shaded in light gray are newly inferred messages using extracted binary tokens.



Request message table				Response message table	
ID	Message	ID	Message	ID	Message
1	GET (argument)	5	OPTIONS (argument)	A	200
2	POST (argument)	6	CONNECT (argument)	B	400
3	PUT (argument)	7	HEAD (argument)	C	404
4	DELETE (argument)	8	TRACE (argument)	D	405

Fig. 5. Inferred HTTP state machine

FTP. For example, in a `vsftpd` state machine, it shows an attempt to set an anonymous login, set the connection port with the `PASV` command, and retrieve a file.

It is noteworthy the result shows that PRETT can infer different types of state machines depending on the implementation even if they implement the same protocol. PRETT inferred a partial state machine for `ProFTPD` compared to that for `vsftpd`. The reason is that the `vsftpd` supported the `RETR` command after an anonymous login, but not `ProFTPD` in our experimental setup.

Simple Mail-Transfer Protocol. The inferred state machine and message types of SMTP are shown in Fig. 4. The experimental result shows that PRETT can infer the process of writing a mail. The process consists of setting both the mail sender and receiver, and composing the mail content. For the inference of the state machine of SMTP, we applied a heuristic. We transmitted a message for the mail content with `<CR><LF>.<CR><LF>` attached after the `DATA` command is transmitted to a server in the previous step. We gained the knowledge that the mail content must end with `<CR><LF>.<CR><LF>` from the response to the message `DATA`, which tells `354 End data with <CR><LF>.<CR><LF>`.

Hyper-Text Transfer Protocol. The inferred state machine and message types of HTTP are shown in Fig. 5. The experimental result of the HTTP state machine inference shows a very simple state machine that has only one state. We conclude that PRETT successfully inferred the HTTP state machine because HTTP is known as a stateless protocol.

4.2 Comparative Evaluation

To show that our inference mechanism outperforms prior works, we compared the number of message types in our state machine with those in the other state machines. We only compared the FTP and SMTP state machines because both protocols are commonly reviewed by previous studies and have considerably complex state machines. For the FTP state machine, we chose the `vsftpd` state machine.

Table 3 shows the result of comparing the state machines that are inferred by the existing studies and PRETT, respectively. As most of the existing papers denote the commands only when describing message types in a state machine without detailed information, we evaluated the performance of message type inference by comparing the number of commands inferred. Consequently, the state machines inferred in our study have the largest number of message types.

Table 3. Comparative result of inferred message types in state machines

Research	Base input	# of message types (FTP)	# of message types (SMTP)
Prospex [7]	App. traces & app	-	6 (Postfix)
Work by Xiao [27]	App. traces & app	11 (unknown)	8 (unknown)
Work by Cho [6]	Input alphabet & app	-	9 (Postfix)
Veritas [24]	Network trace	-	7 (unknown)
ReverX [2]	Network trace	17 (unknown)	-
Work by LaRoche [15]	Known commands & app	19 (vsftpd & ProFTPD)	-
PREUGI [26]	Network trace	18 (unknown)	7 (unknown)
PRETT	Token & trace	22 (vsftpd)	13 (Postfix)

Among them, the proposed mechanism found several new messages which had not been found in previous works. In other words, certain important but rare messages are captured by our mechanism. Such capability is important because malicious applications often use messages that will not come up in a common trace of communication.

4.3 Discovery of Unexpected Behaviors

In most cases, protocol servers respond to the counterpart with proper messages for any message received. Even if the server receives a message that is unimplemented or erroneous, a suitable and reliable disconnection process needs to be done. However, as in the following cases, two cases of unexpected disconnection are observed without receiving any response messages using the inferred state machines. It is noteworthy that the messages that trigger abnormal behaviors are inferred using protocol binary tokens, as shown in Figs. 3 and 4.

Postfix. After the server connection with has been established, it received a `220 2.0.0 Ready to start TLS` response message from the server as we transmitted the `STARTTLS` command to the `Postfix` server. Subsequently, it could not receive any SMTP response message, but only the TCP ACK packet, as we transmitted messages to the server thereafter. We investigated the log of the mail server to determine the reason, and it turned out that the server tried to disconnect the connection abruptly because the communication was not established through the transport layer security (TLS). However, even if the server has been changed to the disconnect state, the server did not transmit any rejection such as a TCP FIN signal or an SMTP response message.

ProFTPD. When we transmitted HTTP commands such as `HEAD`, `OPTIONS`, `PATCH`, `CONNECT`, `PUT`, `DATA`, `DELETE`, `POST`, and `GET`, as well as an SMTP command `MAIL` in all states of the inferred state machine to the `ProFTPD` server, we could not receive any SMTP response message but only a TCP FIN signal. That the server transmits a FIN signal in response to such

command messages indicates that some error handling has been implemented. However, this is in contrast with the case of `vsftpd`, which transmits an SMTP response message with error code 500 when it received any of those messages. Therefore, we conclude that the implementation of error handling on the `ProFTPD` server is more partial than that of `vsftpd`.

5 Limitation

Currently this study is focused on text-based protocols. It is desirable to infer both text-based and binary-based protocols rather than the text-based protocol only because many recent protocols are binary based. However, text-based protocols are still widely used by a large number of users and applications because of their useful characteristics. Also, it is possible for text-based protocols to be reverse engineered in a fully automatic way. We plan to set reverse engineering to the binary-based protocol as the future work. Another limitation is that our approach is not applicable to all the text-based protocols. For example, PRETT cannot be applied to protocols that are not operated by the transmitting/responding communications such as TELNET. TELNET exchanges streams of texts instead of individual messages for different messages. Most of the text-based protocols, however, are designed to be communicated using the transmitting/responding method. In our survey, we found that nine out of ten publicly common text-based protocols operate in such manner. Moreover, if the tokens extracted from the implementation of the protocol are obfuscated, our mechanism is not possible. However, this technique is still useful in case of security analysis in open source projects or when developers are analyzing their own programs. The final limitation is that this technique uses network traces as the input; therefore, the result depends on how sound the messages in the trace are. However, it is not the limitation of only this study because studies of protocol reverse engineering based on network trace have the same limitation.

6 Related Work

For the inference of protocol state machines, several types of approaches have been presented. The first approach is based on network traces, which is static analysis. Veritas [24] and ReverX [2] use a probabilistic approach to generate the protocol state machines. Veritas generates multiple partial finite state machines by merging and simplifying them. ReverX generates protocol state machines by determining the transition probability between clusters after messages are clustered. In addition, PRISMA [14] simulates complete and correct sessions based on the inferred state machine. Another approach for state machine inference is using dynamic analysis. Xiao [27] uses a trial-and-error learning method. This approach targets several text-based protocols using the message field inference method by [5]. Prospex [7] clusters sequential protocol messages and analyzes the correlations among them. It produces not only a concise but also an accurate state machine. Some interactive approaches have also been proposed for

state machine inference. In the case of binary probing, the works by Cho [6] and Zhang [29] identify the potential data message fields by sending and receiving messages to target the binary with query strings. These result in the inference of a state machine with high completeness and extensive coverage compared to other approaches. Using a genetic-algorithm approach, Laroche [15] prepared a great number of message candidates and produced the correct message sequences, which are used to reconstruct a state machine.

Although many prior works have inferred the state machine effectively, it is difficult to infer a state machine with a large number of message types when the inputs are traces or human-provided inputs. As shown from the experimental results herein, they have only inferred a limited number of message types.

7 Conclusion

In this paper, we proposed a novel method to infer a state machine with a large number of message types using tokens extracted from network binaries and network traces. We also proposed an algorithm for state machine minimization using a compatibility test between states. We applied our approach to several text-based protocols that are widely deployed and were able to successfully infer their state machines. The comparative evaluation with other studies shows that our approach can infer minimized state machines while having the largest number of message types. We believe that our proposed mechanism can be effectively used for network protocol analysis.

Acknowledgement. This work was supported by Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIP) (No. R0190-16-2011, Development of Vulnerability Discovery Technologies for IoT Software Security, and No. 2017-0-00184, Self-Learning Cyber Immune Technology Development).

References

1. Pcapr. https://pcapr.net
2. Antunes, J., Neves, N., Verissimo, P.: Reverse engineering of protocols from network traces. In: 2011 18th Working Conference on Reverse Engineering, WCRE, pp. 169–178. IEEE (2011)
3. Borisov, N., Brumley, D., Wang, H.J., Dunagan, J., Joshi, P., Guo, C.: Generic application-level protocol analyzer and its language. In: NDSS (2007)
4. Caballero, J., Poosankam, P., Kreibich, C., Song, D.: Dispatcher: Enabling active botnet infiltration using automatic protocol reverse-engineering. In: Proceedings of the 16th ACM Conference on Computer and Communications Security, pp. 621–634. ACM (2009)
5. Caballero, J., Yin, H., Liang, Z., Song, D.: Polyglot: automatic extraction of protocol message format using dynamic binary analysis. In: Proceedings of the 14th ACM Conference on Computer and Communications Security, pp. 317–329. ACM (2007)

6. Cho, C.Y., Shin, E.C.R., Song, D., et al.: Inference and analysis of formal models of botnet command and control protocols. In: Proceedings of the 17th ACM Conference on Computer and Communications Security, pp. 426–439. ACM (2010)
7. Comparetti, P.M., Wondracek, G., Kruegel, C., Kirda, E.: Prospex: protocol specification extraction. In: 2009 30th IEEE Symposium on Security and Privacy, pp. 110–125. IEEE (2009)
8. Cui, W., Kannan, J., Wang, H.J.: Discoverer: automatic protocol description generation from network traces. In: USENIX Security Symposium, Boston, MA, vol. 14 (2007)
9. Cui, W., Paxson, V., Weaver, N., Katz, R.H.: Protocol-independent adaptive replay of application dialog. In: NDSS (2006)
10. Cui, W., Peinado, M., Chen, K., Wang, H.J., Irun-Briz, L.: Tupni: automatic reverse engineering of input formats. In: Proceedings of the 15th ACM Conference on Computer and Communications Security, pp. 391–402. ACM (2008)
11. Duessel, P., Gehl, C., Flegel, U., Dietrich, S., Meier, M.: Detecting zero-day attacks using context-aware anomaly detection at the application-layer. Int. J. Inf. Secur. **16**(5), 475–490 (2017)
12. Gorbunov, S., Rosenbloom, A.: AutoFuzz: automated network protocol fuzzing framework. IJCSNS **10**(8), 239 (2010)
13. Kohavi, Z., Jha, N.K.: Switching and Finite Automata Theory. Cambridge University Press, Cambridge (2009)
14. Krueger, T., Gascon, H., Krämer, N., Rieck, K.: Learning stateful models for network honeypots. In: Proceedings of the 5th ACM Workshop on Security and Artificial Intelligence, pp. 37–48. ACM (2012)
15. LaRoche, P., Burrows, A., Zincir-Heywood, A.N.: How far an evolutionary approach can go for protocol state analysis and discovery. In: 2013 IEEE Congress on Evolutionary Computation, CEC, pp. 3228–3235. IEEE (2013)
16. Li, Z., et al.: NetShield: massive semantics-based vulnerability signature matching for high-speed networks. In: ACM SIGCOMM Computer Communication Review, vol. 40, pp. 279–290. ACM (2010)
17. Luo, J.Z., Yu, S.Z.: Position-based automatic reverse engineering of network protocols. J. Netw. Comput. Appl. **36**(3), 1070–1077 (2013)
18. Ma, R., Wang, D., Hu, C., Ji, W., Xue, J.: Test data generation for stateful network protocol fuzzing using a rule-based state machine. Tsinghua Sci. Technol. **21**(3), 352–360 (2016)
19. Montague, M.: Text-based internet application protocols (2015). http://www-personal.umich.edu/~markmont/tbiap
20. Newsome, J., Brumley, D., Franklin, J., Song, D.: Replayer: automatic protocol replay by binary analysis. In: Proceedings of the 13th ACM Conference on Computer and Communications Security, pp. 311–321. ACM (2006)
21. Sommer, R., Amann, J., Hall, S.: Spicy: a unified deep packet inspection framework for safely dissecting all your data. In: Proceedings of the 32nd Annual Conference on Computer Security Applications, pp. 558–569. ACM (2016)
22. Somorovsky, J.: Systematic fuzzing and testing of TLS libraries. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, pp. 1492–1504. ACM (2016)
23. Wang, H.J., Guo, C., Simon, D.R., Zugenmaier, A.: Shield: vulnerability-driven network filters for preventing known vulnerability exploits. In: ACM SIGCOMM Computer Communication Review, vol. 34, pp. 193–204. ACM (2004)

24. Wang, Y., Zhang, Z., Yao, D.D., Qu, B., Guo, L.: Inferring protocol state machine from network traces: a probabilistic approach. In: Lopez, J., Tsudik, G. (eds.) ACNS 2011. LNCS, vol. 6715, pp. 1–18. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-21554-4_1
25. Wondracek, G., Comparetti, P.M., Kruegel, C., Kirda, E., Anna, S.S.S.: Automatic network protocol analysis. In: NDSS, vol. 8, pp. 1–14 (2008)
26. Xiao, M.M., Luo, Y.P.: Automatic protocol reverse engineering using grammatical inference. J. Intell. Fuzzy Syst. **32**(5), 3585–3594 (2017)
27. Xiao, M.M., Yu, S.Z., Wang, Y.: Automatic network protocol automaton extraction. In: Third International Conference on Network and System Security, NSS 2009, pp. 336–343. IEEE (2009)
28. Yun, X., Wang, Y., Zhang, Y., Zhou, Y.: A semantics-aware approach to the automated network protocol identification. IEEE/ACM Trans. Netw. (TON) **24**(1), 583–595 (2016)
29. Zhang, Z., Wen, Q.Y., Tang, W.: Mining protocol state machines by interactive grammar inference. In: 2012 Third International Conference on Digital Manufacturing and Automation, ICDMA, pp. 524–527. IEEE (2012)