Reverse Engineering of Protocols from Network Traces

Conference Paper · October 2011

DOI: 10.1109/WCRE.2011.28 · Source: DBLP

João Antunes
University of Lisbon
17 PUBLICATIONS **163** CITATIONS

Nuno Ferreira Neves
University of Lisbon
174 PUBLICATIONS **2,223** CITATIONS

Paulo Veríssimo
University of Lisbon
306 PUBLICATIONS **6,561** CITATIONS

Reverse Engineering of Protocols from Network Traces

João Antunes Nuno Neves Paulo Verissimo
LASIGE, Faculty of Sciences, University of Lisboa, Portugal
{jantunes,nuno, pjv}@di.fc.ul.pt

***Abstract*—Communication protocols determine how network components interact with each other. Therefore, the ability to derive a specification of a protocol can be useful in various contexts, such as to support deeper black-box testing or effective defense mechanisms. Unfortunately, it is often hard to obtain the specification because systems implement closed (i.e., undocumented) protocols, or because a time consuming translation has to be performed, from the textual description of the protocol to a format readable by the tools. To address these issues, we propose a new methodology to automatically infer a specification of a protocol from network traces, which generates automata for the protocol language and state machine. Since our solution only resorts to interaction samples of the protocol, it is well-suited to uncover the message formats and protocol states of closed protocols and also to automate most of the process of specifying open protocols. The approach was implemented in a tool and experimentally evaluated with publicly available FTP traces. Our results show that the inferred specification is a good approximation of the reference specification, exhibiting a high level of precision and recall.**

I. Introduction

Network protocols regulate the communication among entities by defining the syntax and semantics of the messages, and the order in which they need to be exchanged. The ability to obtain a protocol specification can, therefore, play an important role in several contexts. For example, it can help on the implementation of effective defense mechanisms, such as firewalls and intrusion detection systems, that use the specifications of the protocols to accurately identify malicious traffic by performing deep packet inspection [1]. Testing tools can take as input a protocol specification to generate test cases that cover the protocol space for conformance testing [2] or to verify if a server is vulnerable to remote attacks [3].

However, it is typically hard to produce protocol specifications. Closed (or undocumented) protocols require reverse engineering the seemingly arbitrary set of bytes that compose each message in order to determine their meaning and structure. Open protocols, on the other hand, are well documented and their (textual) specification is readily available (e.g., IETF protocols), but obtaining their specification is also difficult and time consuming because developers have to carefully analyze the textual description of the protocol and translate it into the format supported by the tools.

Automatic protocol reverse engineering can address most of these difficulties by deducing an approximate specification of a protocol from information about its operation and with minor assumptions about its structure. In this paper, we present a methodology for automatically inferring the language and state machine of the protocol.This approach constructs two automata (one for the language and the other for the protocol state machine) from the sequences of messages and protocol sessions that were observed in network traces, and then, generalizes and reduces them in order to create a concise specification. The methodology can be used both to extract a specification of closed protocols and to automate most of the manual translation of open protocols. Our solution is focussed on clear-text protocols, often used on network servers. By noticing that many of these server protocols are text-based (e.g., HTTP, SIP, IMAP, FTP, Microsoft Messenger), we decided to explore in our approach the way text fields are usually organized and delimited in a message. We also include in the paper a brief discussion on how to extend the approach to binary-based protocols.

The methodology was implemented in ReverX, a tool that infers the protocol specification from a network trace containing a sample of protocol interactions. An experimental evaluation of the tool was carried out using publicly available network traces, to determine if an inferred specification can capture the main characteristics of a protocol. For this experiment, we chose the FTP protocol for two main reasons. First, it is a non-trivial protocol with a reasonable level of complexity that is well-known to most readers, and therefore, it becomes simpler to provide examples in the text. Second, since FTP is documented in an IETF RFC [4], it facilitates the assessment of the results and allows an intuitive comparison between the inferred automata and the ones manually produced from the textual description. The experiments show that the generated automata can recognize the FTP protocol with a high level of precision, recall, and f-measure, even with training sets with a relatively small number of messages (around 1000 packets).

II. ReverX

A protocol is a set of rules that dictates the communication between two or more entities. It defines *message types* (or *formats*) that are composed of a sequence of fields organized with certain rules and that can take values from a given domain. Therefore, a protocol can been seen as a formal language whose syntax rules are specified through a grammar, describing how symbols of an alphabet can be combined to form valid words. Grammars can be represented by deterministic Finite State Machine (FSM) automata, which are commonly used to describe language recognizers, i.e., computational functions that determine whether a sequence of symbols belongs to the language. Likewise, the network protocol also identifies the order in which the messages can be transmitted while programs execute, and consequently, a FSM can also be utilized to represent the relations among the different types of messages. We call this second automaton the *protocol state machine*. It is thus the goal of our methodology to obtain the specification of

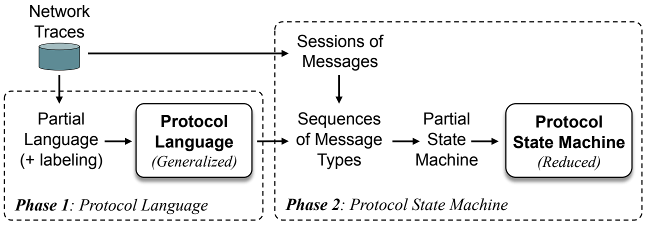

Network Traces
Sessions of Messages
Partial Language (+ labeling)
Protocol Language (Generalized)
Sequences of Message Types
Partial State Machine
Protocol State Machine (Reduced)
Phase 1: Protocol Language
Phase 2: Protocol State Machine

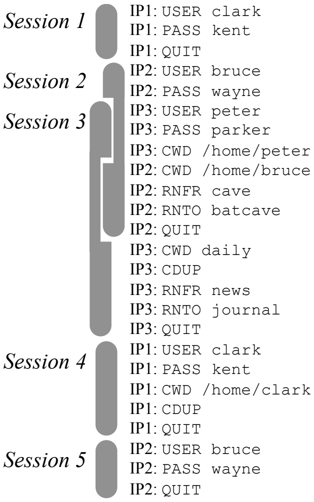
Figure 1. ReverX overview.

the protocol by inferring two FSM, which define the language and the state machine of the protocol.

The problem of grammar inference, or automata induction, refers to the process of learning a language L (or obtaining the FSM that recognizes L) from a set of sample sequences. If the sequences given to infer the language are exhaustive and complete, constructing a FSM that accepts all sequences can be a simple task. However, several problems arise when the language is complex and the alphabet is rich, such as in network protocols. First, if the language defines words that can contain arbitrary values, such a message containing several fields for variable-data arguments, it denotes an alphabet with considerable size (given the combinations of possible values that each field can have). Also, it is quite difficult to produce a representative set of sample sequences for languages with large alphabets, since this would imply that one must obtain a complete set of network traces containing all the messages of the protocol with all possible variations and combinations of parameter data[1]. If the set of sample sequences is incomplete, the problem of grammar inference consists in generalizing from the given set to obtain a FSM that also accepts the missing sequences, without over-generalizing.

Therefore, network protocols are characterized by complex languages and large alphabets, and as a result they are difficult to reverse engineer. Some solutions have nevertheless been created that take advantage of some particular characteristic of the network protocols [5], [6]. Some works also resort to positive and negative examples to learn a grammar from a set of sample sequences [7], [8]. Our approach resorts to network traces that can be easily obtained by intercepting the communication between regular clients and servers and that are expected to contain only positive examples. As in any other learning-based approach, the quality of the derived specification will depend on the correctness and coverage of the sample sequences. Therefore, the network traces should provide a good protocol coverage and must not have any illegal or malformed messages (as they would introduce incorrect message formats or corrupt existing ones). Only messages of the protocol are considered, therefore the traces are previously filtered to select only packets to and from a specific UDP/TCP port number. In addition, to simplify the presentation, we assume that protocol messages are not fragmented in several packets and that no encryption is performed.

Figure 1 depicts an overview of the methodology implemented in the ReverX tool. Since the parties involved in the communication can play distinct roles (e.g., client or server), which restricts the allowed message types that each one can send or receive, we derive separate specifications for each role by looking at the direction of the traffic. The methodology is organized in two phases: In the first phase, ReverX constructs a Prefix Tree Acceptor (PTA) from the protocol messages of the network traces (*Partial Language*), which is then generalized with the intention of producing a FSM that can accept the same message types with different payloads (*Protocol Language*). In the second phase, the tool deduces the protocol state machine from the causal relations among the different messages present in the network traces. ReverX extracts individual protocol sessions, which are converted into sequences of message types, to build a PTA that accepts those sessions (*Partial State Machine*). The tool then resorts to a more aggressive state merging algorithm to reduce the automaton (*Protocol State Machine*).


Session 1
Session 2
Session 3
Session 4
Session 5

```
IP1: USER clark
IP1: PASS kent
IP1: QUIT
IP2: USER bruce
IP2: PASS wayne
IP3: USER peter
IP3: PASS parker
IP3: CWD /home/peter
IP2: CWD /home/bruce
IP2: RNFR cave
IP2: RNTO batcave
IP2: QUIT
IP3: CWD daily
IP3: CDUP
IP3: RNFR news
IP3: RNTO journal
IP3: QUIT
IP1: USER clark
IP1: PASS kent
IP1: CWD /home/clark
IP1: CDUP
IP1: QUIT
IP2: USER bruce
IP2: PASS wayne
IP2: QUIT
```

Figure 2. Example FTP network trace.

A. Inferring the Language

The methodology for deriving the protocol language consists of two parts: the construction of a FSM that accepts only the messages present in the traces, extended with frequency labels, and the generalization of the automaton to accept different instances of the same types of messages. The automaton $L = (Q, \Sigma, \delta, \omega, q_0, F)$ is defined as:

Q is a finite, non-empty set of states,
Σ is the input alphabet, i.e., a finite set of fields extracted from all messages,
δ is the state-transition function: $\delta : Q \times \Sigma \rightarrow Q$,
ω is the labeling-transition function: $\omega : Q \times \Sigma \rightarrow \mathbb{N}$,
q_0 is the initial state, and
F is the set of final states.

In this context, the alphabet of the automaton, i.e., the set of symbols, is the set of message field payloads observed. Transitions from a given state define the message fields that are accepted by that state. Algorithm 1 depicts the method for obtaining the FSM that recognizes the language of the protocol.

1) Construction of the Partial Language Automaton with Frequency Labels: A PTA is built so that it accepts every

[1] This is unreasonable to obtain, for example, in a text field with a variable size of 1 to 256 characters, since there can be over 10^{89} different values.

```
1  Function inferProtocolLanguage
2      Input: $NetworkTraces$ : Messages of the protocol
3          $T_1$ : Minimum ratio of unique instances ($0 \leq T_1 \leq 1$)
4          $T_2$ : Minimum number of transitions ($T_2 > 0$)
5      Output: Automaton $L \leftarrow (Q, \Sigma, \delta, \omega, q_0, F)$
6
7      $q_0 \leftarrow$ NewState()
8      $Q \leftarrow \{q_0\}$
9      $\Sigma \leftarrow \phi$
10     $\delta(q, s) \leftarrow$ UNDEFINED for all domain
11     $\omega(q, s) \leftarrow 0$ for all domain
12     $F \leftarrow \phi$
13     $L \leftarrow (Q, \Sigma, \delta, \omega, q_0, F)$
14
15     // Construction of the partial language (PTA)
16     foreach $m \in NetworkTraces$ do
17         $q \leftarrow q_0$
18         foreach $m_{1 \leq i \leq |m|}$ do // for each message field
19             if $\delta(q, m_i) \neq$ UNDEFINED then
20                 $p \leftarrow \delta(q, m_i)$
21                 $\omega(q, m_i) \leftarrow \omega(q, m_i) + 1$ // inc. frequency
22                 $q \leftarrow p$
23             else
24                 $p \leftarrow$ NewState()
25                 $Q \leftarrow Q \cup \{p\}$ // add new state to Q
26                 $\Sigma \leftarrow \Sigma \cup \{m_i\}$ // add symbol to alphabet
27                 $\delta(q, m_i) \leftarrow p$ // add transition
28                 $\omega(q, m_i) \leftarrow 1$ // initial frequency label
29                 $q \leftarrow p$
30         $F \leftarrow F \cup \{q\}$ // add final state
31
32     // generalize the automaton
33     MinimizeFSM($L$)
34     $generalized \leftarrow$ TRUE
35     while $generalized$ is TRUE do
36         $\Sigma' \leftarrow \{ANY\}$
37         $\delta'(q, s) \leftarrow$ UNDEFINED for all domain
38         $\omega'(q, s) \leftarrow 0$ for all domain
39         $LN \leftarrow (Q, \Sigma', \delta', \omega', q_0, F)$ // nondeterministic FSM
40         $generalized \leftarrow$ FALSE
41         foreach $q \in Q$ do
42             $\#trans_q \leftarrow |\{\delta(q, s) \neq$ UNDEFINED $, s \in \Sigma\}|$
43             $freq_q \leftarrow \sum_{s \in \Sigma} \omega(q, s)$
44             if $\#trans_q / freq_q > T_1$ or $\#trans_q > T_2$ then
45                 // transitions are set with symbol $ANY$
46                 foreach $s \in \Sigma : \delta(q, s) \neq$ UNDEFINED do
47                     $\delta'(q, ANY) \leftarrow \delta'(q, ANY) \cup \{\delta(q, s)\}$
48                     $\omega'(q, ANY) \leftarrow \omega'(q, ANY) \cup \{\omega(q, s)\}$
49                 $generalized \leftarrow$ TRUE
50             else
51                 // transitions keep the same symbol
52                 foreach $s \in \Sigma : \delta(q, s) \neq$ UNDEFINED do
53                     $\Sigma' \leftarrow \Sigma' \cup \{s\}$
54                     $\delta'(q, s) \leftarrow \{\delta(q, s)\}$
55                     $\omega'(q, s) \leftarrow \{\omega(q, s)\}$
56         $L \leftarrow$ DeterminizeFSM($LN$) // converts to deterministic FSM
57         MinimizeFSM($L$)
58     return $L$
```

Algorithm 1. Infer the message formats of the protocol.

message in the network trace (Lines 16–30 in Algorithm 1). The PTA is a FSM where each common prefix of the messages is accepted by the same states and transitions and each unique suffix is accepted by a branchless path in the automaton. We create the PTA from the positive examples of the network trace as follows: Each network message is composed of an arbitrary sequence of bytes, however, text-based protocols usually resort to a fixed delimiter character to separate the message fields, such as a space or a tab. Therefore, by providing a regular expression that defines the field delimiter, we decompose each message as a sequence of fields and separators (abstracted in Line 18). Whenever a symbol (field payload or delimiter) is rejected by the PTA, a new set of states and transitions is added in order to create a path that accepts the entire message. Furthermore, all transitions are labeled with the number of times they were visited (Lines 21–28) to keep track of the frequency that each payload has been observed in that potential field. The resulting frequency-labeled FSM is similar to a probabilistic automaton, where instead of a probability value, each transition has associated an absolute frequency.

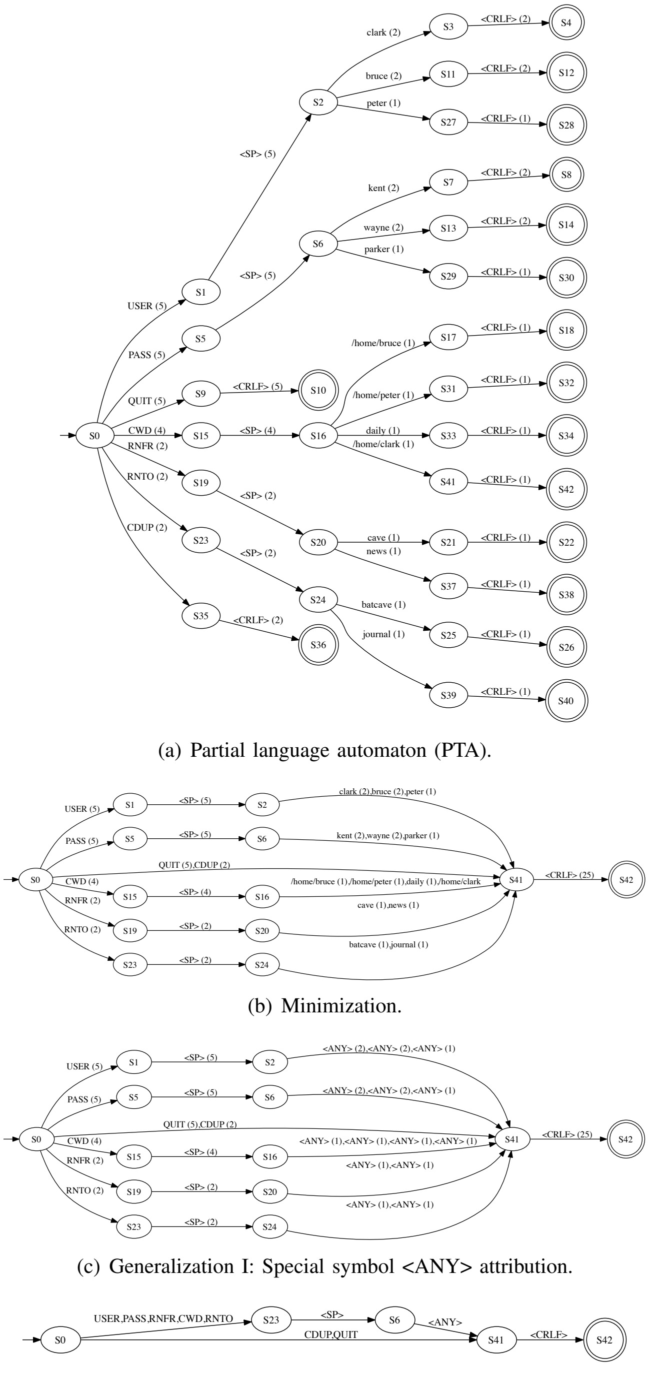
Figure 3 displays a simple example of inferring the protocol message formats from a network trace of Figure 2. The trace was obtained from five simple FTP sessions, just to elucidate the process of deriving a protocol specification using our methodology. To infer the protocol language recognized by the server, the trace was filtered so that it contains only FTP messages sent by the client. Each message is delimited by a carriage return and a line feed characters, and each field is separated by the space character, as specified by RFC 959 [4]. Figure 3(a) shows the partial language PTA derived at the end of this phase, where "<SP>" and "<CRLF>" depict the field separator and message delimiter, respectively. Additionally, each transition is labeled with the frequency that it was visited (in parenthesis).

2) Generalization and Minimization: The partial language PTA is only able to recognize the previously processed messages. In order to produce a more generic FSM that accepts other messages, one needs to identify and abstract the parts of the message format that are not fundamental to the specification (e.g., parameters of a command). At the same time, we also want to produce a concise automaton with a minimal number of states and transitions. Otherwise, the same message fields could be scattered among equivalent states and transitions, needlessly augmenting the complexity of the derived specification.

First, we employ the Moore reduction procedure for deterministic finite automata minimization [9] that produces an equivalent FSM with the minimum number of states and transitions (Line 33). During the minimization process, equivalent states are merged, i.e., states with similar transitions that lead to equivalent states. Merging two states also causes equivalent transitions to be merged and the resulting label is combined by adding the respective frequencies. Figure 3(b) shows the automaton calculated after minimization.

Once the automaton is minimized, we analyze the labelled frequencies to identify parts of the automaton that should be generalized. Our approach is taken from the idea that most protocols make use of the concepts of messages with commands and parameters. To facilitate the parsing of the protocol messages some predefined fields define how each message should be processed, determining the meaning of the remaining bytes. Most textual protocols, for instance, resort to command fields (usually the first) with the command name, usually followed by the respective parameters with variable data (or by some other sub-commands). The different keywords that each command field can have are specified by the protocol and should therefore be inferred. However, the specific parameter data should be abstracted away and generically identified as parameter fields.

Intuitively, fields associated with predefined values, such as command keywords, should appear often in the network traces, as opposed to the variable and less recurrent nature of



(a) Partial language automaton (PTA).

(b) Minimization.

(c) Generalization I: Special symbol <ANY> attribution.

(d) Generalization II: Determinization and minimization.

Figure 3. Inference of the protocol language.

the parameter data. Parameters can therefore be recognized in states of the automaton that accept a wide range of different symbols (each one is a particular instance of that parameter). Additionally, these symbols should appear with relatively low frequency, since each individual instance of a parameter should be much less common than a command keyword. Alongside, one can not rely only on the individual frequency of each symbol, or else commands that appear rarely in the traces could be misidentified as parameters.

The part of the algorithm responsible for identifying the states that should be generalized appears in Lines 34–57. Two configuration parameters are employed T_1 and T_2 (Line 44). Any state of the automaton is selected for generalization if one of two conditions is satisfied:

C1) the ratio of the number of symbols recognized by a state over the total frequency of that state is above T_1;
C2) the total number of symbols is larger than T_2.

The condition on T_1 determines that a field is a parameter by looking for states that accept a wide range of symbols relative to the total number of times they were observed in the traces (i.e., the sum of frequency labels on that state). Therefore, T_1 must be set to a value that captures the variability and sporadic nature of parameters. Consider for example a message field that can hold four distinct command names. In 200 messages, the value of the field will be distributed among those four commands (not necessarily evenly), and therefore the ratio of symbols over the total frequency will be $4/200 = 0.02$. On the other hand, a state that represents a parameter field (e.g., a pathname) is not bound to a limited number of fixed symbols. On 200 messages, the field could have 150 different values and the ratio would be $150/200 = 0.75$. The evaluation section studies the sensitivity of the methodology to T_1, and it is possible to observe that generalization works effectively for a wide range of values.

The condition on T_2 says that every state accepting more symbols than what a typical command field would, should also be considered a parameter. The purpose of this condition is to address traces that are heavily skewed toward a certain command (or commands), making its parameters appear unusually common, and possibly causing them to be incorrectly regarded as commands. Therefore, T_2 is quite generic and only needs to be greater than the maximum number of different commands that any protocol field can have (e.g., $T_2 = 30$ is acceptable because it is unlikely for a command field to accept more than 30 different command names).

The states identified as field parameters are generalized by making each transition leaving from those states accept any value (special symbol ANY). This transformation however may render the FSM nondeterministic, which we denote by automaton LN. The LN automaton is constructed from L (Lines 36–39) and each state is evaluated and subject for generalization (Lines 41–55). At the end of this stage, we employ a standard determinization algorithm [9] to merge all nondeterministic transitions (i.e., those with symbol ANY) into a single transition, effectively producing a new generalized version of the language FSM L (Line 56). The minimization algorithm is again applied to produce a simpler, yet equivalent, automaton (Line 57). This procedure is repeated until no more states can be generalized.

Returning to the FTP example, Figure 3(c) represents the generalized nondeterministic automaton LN after the first iteration of the loop. ReverX was configured with $T_1 = 0.4$ and $T_2 = 30$, and state $S0$ was not generalized ($7/25 = 0.28 < T_1$) while states S2 and S6 were identified as parameters ($3/5 = 0.6 > T_1$). The result of converting the nondeterministic automaton to a deterministic one, followed by the minimization operation is shown in Figure 3(d). This automaton also corre-

```
1  Function inferStateMachine
2      Input: $Sessions$ sequences of message formats
3      Output: Automaton $S \leftarrow (Q, \Sigma, \delta, q_0, F)$
4
5      $q_0 \leftarrow$ NewState()
6      $Q \leftarrow \{q_0\}$
7      $\Sigma \leftarrow \phi$
8      $\delta(q, s) \leftarrow$ UNDEFINED for all domain
9      $F \leftarrow \phi$
10     $S \leftarrow (Q, \Sigma, \delta, q_0, F)$
11
12     // Construction of the partial state machine (PTA)
13     foreach $sx \in Sessions$ do
14         $q \leftarrow q_0$
15         foreach $m_{1 \leq i \leq |sx|} \in sx$ do // for each message type
16             if $\delta(q, m_i) \neq$ UNDEFINED then
17                 $q \leftarrow \delta(q, m_i)$
18             else
19                 $p \leftarrow$ NewState()
20                 $Q \leftarrow Q \cup \{p\}$ // add new state to Q
21                 $\Sigma \leftarrow \Sigma \cup \{m_i\}$ // add message type to alphabet
22                 $\delta(q, m_i) \leftarrow p$ // add transition
23                 $q \leftarrow p$
24         $F \leftarrow F \cup \{q\}$ // add final state
25
26     // merge states reached from similar message types
27     foreach $q \in Q$ do
28         foreach $p \in Q$ do
29             if $\exists s \in \Sigma; r, t \in Q : \delta(q, s) = r \wedge \delta(p, s) = t$ then
30                 MergeStates($\delta(q, s), \delta(p, s)$)
31
32     // merge states without a causal relation that share at least
33     // one message type
34     $reduce \leftarrow$ TRUE
35     while $reduce$ is TRUE do
36         $reduce \leftarrow$ FALSE
37         foreach $q \in Q$ do
38             foreach $p \in Q$ do
39                 // if there is not a casual relation
40                 if $(\nexists s \in \Sigma : \delta(q, s) = p \vee \delta(p, s) = q)$ or
41                    $(\exists s, t \in \Sigma : \delta(q, s) = p \wedge \delta(p, t) = q)$ then
42                     if $\exists s \in \Sigma; r \in Q : \delta(q, s) = r \wedge$
43                                         $\delta(p, s) = r$ then
44                         MergeStates($p, q$)
45                         $reduce \leftarrow$ TRUE
46         MinimizeFSM($S$)
47     return $S$
```

Algorithm 2. Infer the state machine of the protocol.

sponds to the final FSM for the language, since the second loop iteration did not find any more states to generalize.

B. Inferring the State Machine

A protocol specification also defines the casual relations between messages. In the second phase, the methodology uses the traces and the previously inferred language to obtain the protocol state machine. Algorithm 2 presents the method used to get the state machine of the protocol.

1) Extracting the Application Sessions: The protocol state machine automaton is constructed from the application *sessions*, each one corresponding to a sequence of messages exchanged during the same interaction between both parties. To identify individual sessions in the traces, we cluster messages that share similar network characteristics, such as network addresses and time proximity. In the current version, ReverX groups each application session based on the following criteria:

- same source and destination IP and port addresses;
- TCP sequence numbers follow a monotonic increasing function, with the next sequence number being at most the sum of the last sequence number with the length of that last packet;
- temporal gaps between messages smaller than one hour.

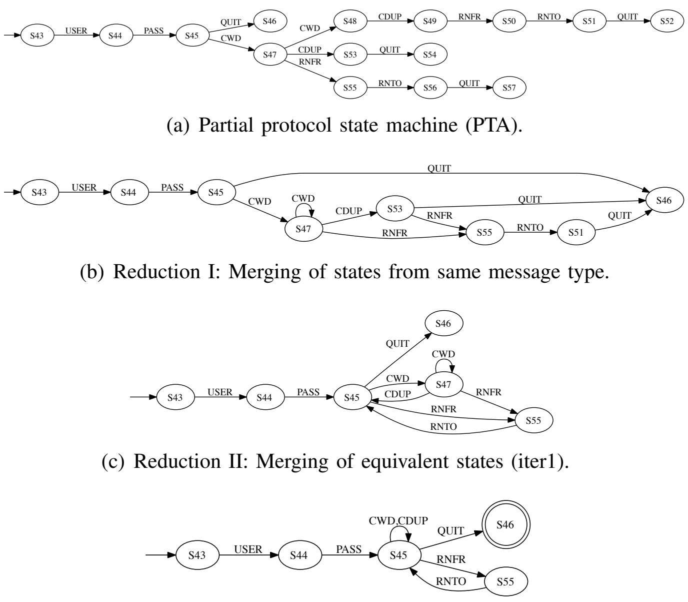
Then, we use the inferred language from the first phase to convert each session into a sequence of message types. Every path in the language automaton corresponds to a distinct message format, and it receives a unique identifier naming the specific message type. Therefore, one can determine the type of a message by processing it with the language FSM. By following this approach iteratively, ReverX transforms each session as a sequence of message type identifiers.

2) Construction of the Partial State Machine Automaton: Analogous to the protocol language inference, we build a PTA that accepts the sequences of message types present in the application sessions (Lines 13–24). Since frequency information is not needed by Algorithm 2, the automaton S does not define a labeling function. New states and transitions are added to the automaton whenever a distinct message type appears. In the end, the automaton is able to recognize all sessions observed in the network traces.

Figure 4 exemplifies how the partial protocol state machine is inferred from the network trace and from the derived language FSM. After clustering the network messages into individual protocol sessions, as depicted in Figure 2, the tool converts the sessions into sequences of message types. The FSM built from those sequences appears in Figure 4(a). For the sake of readability, each message type is identified with the name of the command of the message.

3) Reduction: The current FSM only captures the sequence of transitions between the protocol messages exactly as they appear in the traces. To derive the protocol state machine, it is necessary to identify and merge the automaton states that correspond to the same protocol state. In the first place, we find out which states are reached under similar conditions, i.e., from the same message type, because they probably represent the same protocol state. Following this idea, the algorithm merges all destination states of transitions that define the same symbol (Lines 27–30). The merge operation consists in: creating a new state with transitions from each pair of states to merge, removing these two states from the automaton, and updating any transition that pointed to either of these states. As with most heuristic approaches, extrapolating from a smaller subset may incur in over-optimistic lossy generalizations. For instance, this procedure may fail if the protocol specification defines the same message type (e.g., the same command name) for different purposes. However, we found this practice to be very uncommon in the usual protocol specifications. Figure 4(b) shows the automaton after this procedure.

However, some states that are reached from different message types may be part of the same protocol state. For instance, after logging in, a user may create, edit, or delete files, all seemingly interchangeable protocol commands. With respect to the protocol state machine, the order of these messages is irrelevant as they are executed from the same state. However, network traces are most probably incomplete, in the sense that many causal relations between protocol messages may be absent. To deduce a complete protocol state machine, in spite of the incompleteness of the network traces, the algorithm needs to



(a) Partial protocol state machine (PTA).

(b) Reduction I: Merging of states from same message type.

(c) Reduction II: Merging of equivalent states (iter1).

(d) Reduction II: Merging of equivalent states (iter2).

Figure 4. Inference of the protocol state machine.

make a few assumptions about the equivalence of some states. First, if there is a transition from a state $S1$ to state $S2$, but not vice versa, then they can never be considered equivalent because it establishes a causal relation from $S1$ to $S2$. Second, protocol states without a direct causal relation (i.e., without any transition between them or with transitions between them in both directions) and without similar transitions (i.e., not accepting at least one common message type), are also never considered equivalent. These two conditions are enforced by the algorithm to determine the cases where states can be merged (Lines 37–44). The algorithm also minimizes the produced FSM to obtain a simpler but equivalent automaton (Line 46). This procedure is repeated until no more states can be merged, and the resulting automaton is the protocol state machine.

Figure 4(c) displays the automaton S after the first iteration of this merge procedure (*while* loop at Line 35). States $S43$ and $S44$, for instance, have a causal relation, and therefore cannot be merged. However, states $S45$, $S51$, and $S53$, are merged because they all share message type QUIT and do not have any causal relation between them. In the second iteration, the tool merges $S45$ and $S47$ that share message type RNFR, and the result is the final protocol state machine (see Figure 4(d)).

C. Extensions for Binary-based Protocols

Our inference methodology is primarily focused on text-based protocols. However, we have some ongoing work on extending our current approach to support binary-based protocols by taking into consideration the characteristics of this kind of network protocols. The fundamental difference between text- and binary-based protocols resides in their language, i.e., the message formats. Text-based protocols are built around the notion of message fields encoded with text data and separated by known characters, whereas binary-based protocols usually resort to fixed-length fields or to a special notation to indicate the length of variable fields. On the other hand, the specification of the state machine is the same across text- or binary-based protocols. Once the language is inferred, the message types, either encoded with text or binary data, are used by the clients and servers to make progress and to jump to the different states of the protocol. Therefore, we only need to extend our approach for the inference of the language of the protocol (the first phase of the methodology).

Three changes can be made to assist the derivation of binary message formats. The first is related with the construction of the PTA, where each protocol message of the trace is decomposed in a sequence of fields (Line 16 in Algorithm 1). In our original approach, we resort to a regular expression that defines the fixed delimiter character(s) used to separate each (text) field. However, because there is no *a priori* assumption about the structure of the binary fields, we divide each message initially as a sequence of fields of just one byte (regular expression "."). Although, one bit is the smallest binary unit, representing fields as one-bit long would result in a very large PTA and would greatly impact the overall performance of the inference process. Moreover, fields that are smaller than one byte can also be derived, although concatenated with other small fields that together are inferred as a larger one-byte field. Most binary protocols usually group several one-bit fields (i.e., flags) into blocks of one, two, or four bytes, due to performance issues.

Once the PTA is constructed, each transition deemed as a message field is defined with a one-byte long symbol due to the message decomposition. However, if a particular subpath in the automaton is made up of branchless transitions, i.e., states with only one transition, they can be concatenated into a single state and transition, thus defining larger message fields. We thus introduce the second modification to the methodology that concatenates all symbols from branchless subsequences of states and transitions (after Line 33). The concatenation works as follows: for every three states $q_0, q_1, q_2 \in Q$ that define only two transitions $\delta_0(q_0, a) = q_1$ and $\delta_1(q_1, b) = q_2$, with $a, b \in \Sigma$, we merge states q_0 and q_1 and replace both transitions with $\delta_2(q_{01}, ab) = q_2$. We also can use this concatenation function in the end of each generalization cycle (after Line 57), mostly for readability purposes but also to minimize the number of states and transitions in the final automaton.

The third and final modification is related with the derivation of the size of binary fields. Once a state is selected for generalization (Lines 46–49), instead of replacing the symbol of each transition with the special symbol "<ANY>" (which in text-based protocols can be implemented as the regular expression ".+" or "[a-zA-Z0-9]+"), we now keep information about the length of the original symbol, such as ".{4}+" for a field that defines a generic symbol 4 bytes-long.

An initial evaluation with traces of the Domain Name System (DNS) protocol show that our approach does produce a correct automaton that accepts all message types from the traces, but the generalization was only accomplished to some extent (the initial header of the messages). This level of generalization can be acceptable to some contexts (e.g., firewall rules), but further research is still needed to improve and refine the algorithm.

III. Evaluation

This section evaluates ReverX to assess the quality of the inferred language and state machine automata for a given protocol. To attain this objective, we chose to derive a specification of the FTP protocol because it has a reasonable level of complexity. Since FTP is documented by a IETF RFC [4], it allows the comparison between the inferred automaton and a reference automaton, manually produced from the documentation. The network traces were obtained from a public repository to facilitate the reproducibility of the results and to demonstrate that our solution can use unbiased network traces obtained from the regular utilization of the protocol (without being specifically produced for reverse engineering purposes).

A. Experimental Framework

1) FTP Protocol: To assess the quality of our inference methodology, we reverse engineer the File Transfer Protocol (FTP) [4]. FTP defines a standard way where clients can access files stored remotely in a server. The complete RFC 959 specification defines 33 commands, allowing clients to authenticate, download or upload files, create directories, delete or rename files or directories, or to obtain status information about files and directories.

2) Network Traces: The evaluation uses publicly available FTP network traces[2]. Even though these traces had been previously processed to anonymize the clients [10], a few packets still contained malformed messages that had to be cleaned (such as illegal command names). Furthermore, we chose to concentrate on the RFC 959 specification [4], hence we filtered out any network messages non compliant with this standard. This resulted in a clean packet capture file containing 868 825 FTP messages (out of the original 886 547 messages).

3) Evaluation Methodology: The evaluation focuses on deriving the client side of the FTP protocol (an equivalent approach could be utilized for the server side), and therefore only the client FTP messages were used in the experiments. Overall, 10 independent experiments were conducted, each one employing a subset of the trace as training data and the remaining messages as test sets. In more detail, the procedure for each experiment is: First, we randomly select a point in the trace to pick 4000 consecutive FTP messages as the training set. ReverX infers the language and state machine of the protocol for various configurations based on this set, by varying the sample size (raging from 250 messages to the whole 4000) and the generalization parameter T_1 (from 0.0 to 1.0). Therefore, two automata are produced for each configuration, totaling 132 FSM per experiment. Then, each automaton is evaluated using 10 test sets, which are generated by randomly choosing 4000 consecutive messages from the remaining trace file.

We assess the quality of the inferred automata by calculating the following metrics:

- *Recall*: measures the coverage of the inferred automaton, i.e., how much of the protocol specification has been captured by the FSM. Recall is calculated as the ratio that a randomly selected set of *valid* protocol messages (or protocol sessions) is accepted by the inferred automaton.

$$\mathit{Recall} = \frac{\text{\# accepted messages (or sessions)}}{\text{\# messages (or sessions)}}$$

- *Precision*: determines the soundness of the automaton, i.e., if the inferred automaton is not overly-generalized. We calculate precision as the ratio that a randomly selected (*valid* or *invalid*) set of protocol messages (or sessions) accepted by the inferred FSM is in fact *valid*.

$$\mathit{Precision} = \frac{\text{\# accepted } valid \text{ messages (or sessions)}}{\text{\# accepted messages (or sessions)}}$$

- *F-score*: measures the accuracy of a test by computing a score that considers the precision and the recall.

$$\mathit{F\text{-}Score} = 2 \times \frac{\mathit{Precision} \times \mathit{Recall}}{\mathit{Precision} + \mathit{Recall}}$$

Typically, it is very simple to have a recall of 1 (with a low precision) by having a very generic automata that accepts all messages (or sessions). Therefore, our goal is to achieve both a high recall *and* precision [11], which is reflected by f-score values close to 1.

4) Testbed: The experiments were carried out in a Intel Pentium Dual Core 2.8GHz with 2GB of memory running Ubuntu 9.04. ReverX is programmed in Java and resorts to libpcap[3] and jnetpcap[4] libraries to access packet capture files in TCPDUMP format. ReverX also uses the dot program[5] to generate high-quality diagrams of the automata.

B. Experimental Results

All experiments described in this section were executed with $T_2 = 30$. Each entry of recall, precision, and f-score is an average of 100 values (10 experiments, each with 10 different test sets).

1) Protocol Language: To calculate the recall of the FSM of the language, we used the 4000 messages of each test set in the inferred automaton to find out which packets were accepted or rejected. A recall with a value near 1 means that most of the messages are recognized and, therefore, that the FSM should be able to capture most of the protocol language used by the FTP clients.

To calculate the precision, we require messages that are accepted by the derived automaton and evaluate if they are accepted or rejected by the reference language of FTP. This gives us a measure of how accurate is the inferred automaton, since a higher precision value indicates that fewer extraneous messages are recognized. To get data for the experiment, we decided to follow an approach based on the mutation of test set messages, to produce (mutated) messages that are still accepted by the inferred automaton but could potentially be rejected by the reference automaton. We configured the `editcap` tool[6] to mutate each byte of every packet with a probability of 0.1. Then, we fed the mutated messages to the inferred FSM and only kept the messages that were accepted. This process was applied repeatedly to the original test set, until a mutated (but accepted) test set was produced that contained 4000 messages.

[2] http://ee.lbl.gov/anonymized-traces.html

[3] http://www.tcpdump.org/

[4] http://jnetpcap.com/

[5] http://www.graphviz.org/

[6] http://www.wireshark.org/docs/man-pages/editcap.html

Table I
EVALUATION OF THE INFERRED SPECIFICATION.

(a) Language FSM evaluation.

T_1		Training set size (messages)					
		250	500	1k	1.5k	2k	4k
0.00	Prec.	1.00	1.00	1.00	1.00	1.00	1.00
	Rec.	0.58	0.58	0.58	0.58	0.58	0.58
	F-Sc.	0.73	0.73	0.73	0.73	0.73	0.73
0.10	Prec.	0.81	0.82	0.99	1.00	1.00	1.00
	Rec.	1.00	1.00	1.00	1.00	1.00	1.00
	F-Sc.	0.89	0.90	1.00	1.00	1.00	1.00
0.20	Prec.	0.39	0.75	0.99	1.00	1.00	1.00
	Rec.	1.00	1.00	1.00	1.00	1.00	1.00
	F-Sc.	0.56	0.86	1.00	1.00	1.00	1.00
0.30	Prec.	0.39	0.75	0.99	1.00	1.00	1.00
	Rec.	1.00	1.00	1.00	1.00	1.00	1.00
	F-Sc.	0.56	0.86	1.00	1.00	1.00	1.00
0.40	Prec.	0.39	0.75	0.99	1.00	1.00	1.00
	Rec.	1.00	1.00	1.00	1.00	1.00	1.00
	F-Sc.	0.56	0.86	1.00	1.00	1.00	1.00
0.50	Prec.	0.39	0.75	0.98	1.00	1.00	1.00
	Rec.	1.00	1.00	1.00	1.00	1.00	1.00
	F-Sc.	0.56	0.85	0.99	1.00	1.00	1.00
0.60	Prec.	0.39	0.75	0.98	0.99	1.00	1.00
	Rec.	1.00	1.00	1.00	1.00	1.00	1.00
	F-Sc.	0.56	0.85	0.99	1.00	1.00	1.00
0.70	Prec.	0.39	0.74	0.98	0.99	1.00	1.00
	Rec.	1.00	1.00	1.00	1.00	1.00	1.00
	F-Sc.	0.56	0.85	0.99	0.99	1.00	1.00
0.80	Prec.	0.38	0.73	0.97	0.97	0.98	0.99
	Rec.	1.00	1.00	1.00	1.00	1.00	1.00
	F-Sc.	0.55	0.84	0.98	0.98	0.99	0.99
0.90	Prec.	0.38	0.72	0.95	0.96	0.97	0.98
	Rec.	1.00	1.00	1.00	1.00	1.00	1.00
	F-Sc.	0.55	0.84	0.97	0.98	0.99	0.99
1.00	Prec.	0.34	0.71	0.94	0.96	0.97	0.98
	Rec.	1.00	1.00	1.00	1.00	1.00	1.00
	F-Sc.	0.50	0.83	0.97	0.98	0.98	0.99

(b) State machine FSM evaluation.

T_1		Training set size (messages)					
		250	500	1k	1.5k	2k	4k
0.00	Prec.	0.00	0.00	0.00	0.00	0.00	0.00
	Rec.	0.00	0.00	0.00	0.00	0.00	0.00
	F-Sc.	N/A	N/A	N/A	N/A	N/A	N/A
0.10	Prec.	0.58	0.70	0.98	0.98	0.98	1.00
	Rec.	0.99	0.99	1.00	1.00	1.00	1.00
	F-Sc.	0.73	0.82	0.99	0.99	0.99	1.00
0.20	Prec.	0.60	0.70	0.98	0.98	0.98	1.00
	Rec.	0.99	0.99	1.00	1.00	1.00	1.00
	F-Sc.	0.75	0.82	0.99	0.99	0.99	1.00
0.30	Prec.	0.61	0.68	0.98	0.98	0.98	1.00
	Rec.	0.99	0.99	1.00	1.00	1.00	1.00
	F-Sc.	0.75	0.81	0.99	0.99	0.99	1.00
0.40	Prec.	0.56	0.68	0.98	0.98	0.98	1.00
	Rec.	0.99	0.99	1.00	1.00	1.00	1.00
	F-Sc.	0.71	0.81	0.99	0.99	0.99	1.00
0.50	Prec.	0.56	0.68	0.97	0.98	0.98	1.00
	Rec.	0.99	0.99	1.00	1.00	1.00	1.00
	F-Sc.	0.71	0.80	0.99	0.99	0.99	1.00
0.60	Prec.	0.56	0.67	0.97	0.98	0.98	1.00
	Rec.	0.99	0.99	1.00	1.00	1.00	1.00
	F-Sc.	0.71	0.80	0.99	0.99	0.99	1.00
0.70	Prec.	0.56	0.68	0.97	0.97	0.98	1.00
	Rec.	0.99	0.99	1.00	1.00	1.00	1.00
	F-Sc.	0.71	0.81	0.99	0.98	0.99	1.00
0.80	Prec.	0.56	0.69	0.97	0.95	0.96	0.99
	Rec.	0.99	0.99	1.00	1.00	1.00	1.00
	F-Sc.	0.71	0.81	0.99	0.97	0.98	0.99
0.90	Prec.	0.56	0.68	0.94	0.93	0.94	0.98
	Rec.	0.99	0.99	1.00	1.00	1.00	1.00
	F-Sc.	0.71	0.81	0.97	0.97	0.97	0.99
1.00	Prec.	0.47	0.66	0.93	0.93	0.94	0.98
	Rec.	0.99	0.99	1.00	1.00	1.00	1.00
	F-Sc.	0.64	0.79	0.96	0.97	0.97	0.99

To finalize the calculation of precision, we resorted to the reference FSM of FTP to verify which messages of the mutated test set were in fact legal.

Table I(a) shows the recall and precision of the protocol language automata inferred by ReverX (grey cells have recall and precision values over 0.9). For each value of the generalization parameter T_1, we produced FSMs from different sizes of the training set (ranging between 250 to 4000 messages). It is possible to conclude from the table that smaller training set sizes lead to worse quality automata. This is expected since smaller traces lack enough message diversity to make them representative enough of the protocol language[7]. On the other hand, it is possible to see that even for relatively small training sets (e.g., 1000 messages) one can already infer high quality FSMs. Of course, in general there is no exact number for the minimum training set size because it depends on both the protocol complexity and coverage of the trace.

The generalization parameter also affects the quality of the automata. For instance, a value of 0.0 results in generalizing every parameter as a state, therefore producing over-generalized automata, which accounts for a recall of 1.0 and the lowest precision values. This kind of over-generalized FSM recognizes any type of FTP message, but it also accepts illegal messages. On the other extreme of the spectrum, a generalization parameter of 1.0 creates FSMs that never generalize. These automata reject some legal FTP messages but they are unlikely to accept any illegal messages. In any case, ReverX seems to be relatively insensitive to T_1, since it produces good results for a large range of generalization values.

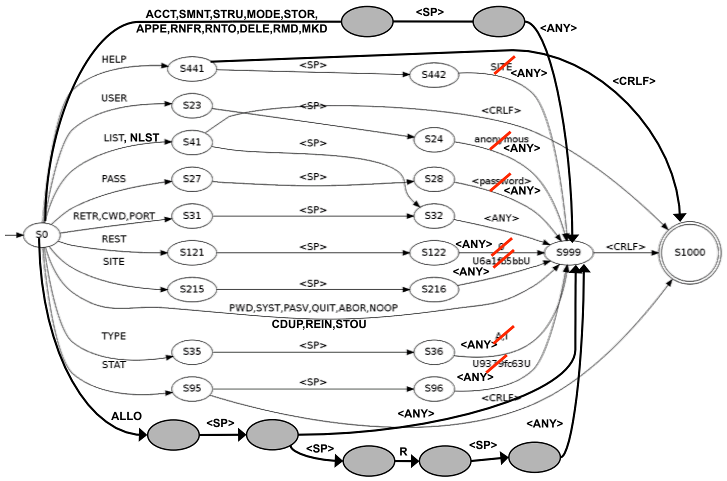


Figure 5. Inferred protocol language versus RFC 959.

Figure 5 compares the best derived automaton (with training set size of 4000 and generalization parameter of 0.3) against the reference FSM for FTP. The gray states and bold lines and labels correspond to states and transitions present in RFC 959, but not inferred by ReverX. The partial inference of the message formats is due to a generic limitation of trace-based solutions—they can not infer what is absent from the traces. In fact, all missing commands are not present in the network trace (e.g., ALLO, ACCT, CDUP). Additionally, some transitions are not generalized because of the low diversity of command parameters present in the traces (e.g., "SITE U6a1fb5bbU") or due to the anonymization procedure (e.g., "USER anonymous").

2) Protocol state machine: The generated protocol state machines are also evaluated with similar metrics. Recall is obtained using the protocol sessions extracted from the test sets, which are then tried in the inferred FSMs. To calculate the precision, we employ an equivalent approach where the extracted protocol sessions are mutated with probability 0.1. Here, the mutation simply consists in either deleting the message or in swapping a given message with the one immediately succeeding it. This simple method is a very convenient to create potentially invalid protocol sessions that could be accepted by an over-generalized FSM. For instance, if a USER command must always precede a PASS command, a mutation on the first message would effectively render the session invalid. To verify if the sessions accepted by the inferred FSM are in fact valid, we built a reference FSM for the RFC 959 that only recognizes legal FTP protocol sessions.

TableI(b) shows the precision, recall, and f-score values for the obtained protocol state machines. As with the protocol language inference, the best FSM are built from larger samples of the training set (i.e., 1000 messages or more). Here, the impact of the sample size is more pronounced because smaller training sets have fewer protocol sessions. In fact, we calculated an average of 29 messages per session, and therefore, a set with 250 messages would only contain around 8 sessions, which is typically insufficient to generate a good FSM. The results also show that 1000 messages (i.e., an average of 34.5

[7] In fact, more than 70 percent of the messages had the same kind of protocol request (PORT command) in the trace that was used.

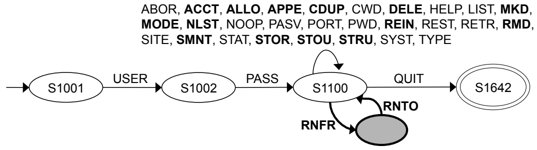

ABOR, ACCT, ALLO, APPE, CDUP, CWD, DELE, HELP, LIST, MKD,
MODE, NLST, NOOP, PASV, PORT, PWD, REIN, REST, RETR, RMD,
SITE, SMNT, STAT, STOR, STOU, STRU, SYST, TYPE
S1001
USER
S1002
PASS
S1100
QUIT
S1642
RNTO
RNFR

Figure 6. Inferred protocol state machine versus RFC 959.

sessions) are nevertheless sufficient to create FSMs that capture the state machine of the protocol. The other factor affecting the inference is the generalization parameter used to obtain the language FSM. This is clearly seen in the recall and precision scores for T_1 equal to 0.0, which results in over-generalized language FSMs and in an inferred protocol state machine with a single state that rejects all FTP protocol sessions (low recall and precision values). Overall, the results show that ReverX is able to obtain a protocol state machine from a relatively small sample of the network trace (i.e., 1000 messages) and using a reasonable interval of generalization values (i.e., between 0.1 and 0.8).

The best derived state machine (with training set size of 4000 and the inferred language FSM with generalization parameter of 0.3) and the reference state machine are depicted in Figure 6. The gray state and transitions and labels in bold correspond to the part of the FTP specification not captured by ReverX. As before, ReverX only failed to infer protocol states missing from the traces. Nevertheless, ReverX is able to deduce all states observed in the traces, such as the USER and PASS commands issued before any others, and the QUIT command as the final state. Even though the RNFR and RNTO commands are missing from the network traces, our approach is able to correctly identify those states if they are present in the network traces, such as shown in Figure 4.

IV. Related Work

In this paper, we present a solution for inferring a protocol specification based on automata generation from a training set. The problem of automata inference has been tackled in different research areas in the past, from natural languages to biology and to software component behavior [12]–[14]. Typically, a prefix tree acceptor is first built from the training set, accepting all events. Then, similar states are merged according to their local behavior (e.g., states with the same transitions or states that accept the same k consecutive events) [13], [15]. Some solutions also resort to specific rules or heuristics to aid the inference of the automata [16]. Although these techniques can produce useful models, their precision can be affected when dealing with larger and complex models, and some works have tried to addressed this limitation [17].

Protocol reverse engineering has been traditionally a laborious and manual task, with a few tools to ease the process of capturing and analyzing individual network packets [18]–[21]. It was only recently that the field of automatic inference of protocol specifications has seen some developments. The great majority of these works focused on the inference of the protocol language, i.e., they try to derive the message formats accepted by the protocol. Two distinct approaches have been applied—study the dynamics of a program that implements the protocol, and resort to the analysis of the network traffic generated between parties.

Dynamic analysis tools closely monitor the program's execution while processing a single message. Taint analysis is employed to identify the code that parses the packets, and to correlate it with each part of the message. The resulting execution trace is then examined to locate message fields and their content type (e.g., length) [22]–[25]. Even though these tools have shown interesting practical results, they can suffer from some limitations. For instance, if the server employs non-standard libraries or if its parsing mechanisms deviate from what is expected, dynamic analysis tools may be unable to make any sense of the fields or even the entire message. Additionally, the use of techniques for software piracy prevention, such as obfuscation [26], can preclude the understanding of the code. These tools are also system and programming language dependent, due to the taint analysis engine, which constrains the programs that can be analyzed.

A few works have attempted to infer parts of the protocol language from network traces. Protocol Informatics employs bioinformatics sequence alignment algorithms to reveal similarities between messages, and then consensus sequences are studied to find the location and lengths of some message fields [5]. Discoverer resorts to a different approach to derive more information about the messages [6]. It uses an initial clustering to group messages with similar sequences of text or binary tokens, and then, recursive clustering and sequence alignment to refine each cluster and produce more detailed message formats. Experiments have shown that Discoverer could not correctly infer about 10% of the message formats, in part due to some inaccurate parsing.

We are aware of three approaches to derive the state machine of the protocol. Prospex employs taint analysis to obtain execution traces for each execution session, which are then used to build an acceptor machine [27]. Message formats are inferred with equivalent techniques as [25]. The state machine is generated by building an augmented prefix tree from the sequences of message types of the sessions, and then by transforming the tree into the smallest automaton that is consistent with the training data. However, the taint analysis used by Prospex suffers from similar limitations as above, such as requiring a controlled environment to run and to collect the program execution data. PEXT utilizes network traces to infer an approximate state machine [28]. First, it clusters messages based on a distance metric using the length of the longest common substring and labels each message with the corresponding cluster ID. Then, it translates each session into a sequence of cluster IDs. States and transitions are generated from similarities between the sequences of IDs in the sessions and the order in which they appear in the traces. This approach is useful to evidence patterns of sequences of messages that arise from using specific protocol features. However, it can not derive the message formats, creating a semantic gap between the final automaton and the observed data. The clustering method can be error prone because the use of the longest common substring metric might induce incorrect clustering of different message types that share long common parameters (e.g., path name). Trifilo et al. describe protocol reverse engineering solution that resorts to the statistical analysis of network traces [29]. This approach however assumes a single message format for the protocol, which allows all messages to be aligned and compared. The distributions of the variance of the bytes over different messages are compared in order to identify the most relevant field, i.e., the field that is most likely to dictate the logic of the protocol. The protocol state machine is then obtained from the order of messages in the traces and the values of this relevant field. While this may provide good results for some binary protocols (ARP), it is not suitable for the majority of application protocols because they have different message formats (e.g., FTP or IMAP). In addition, it may be insufficient to use the variance of distributions as a mean to detect the most relevant field(s), as the results are greatly dependent on how uniformly the various kinds of messages appear in the traces[8].

In this work, we describe and evaluate a new methodology for deriving a concise representation of a protocol, both for message formats and state machine. Our approach is solely based on traces, which may be readily available in the web or can be easily obtained by collecting network traffic. At this stage, we are focusing text-based protocols, which are commonly used by client-server applications.

V. Conclusions and Future Work

This paper presents a new methodology and a tool to derive a protocol specification from network traces. Our approach resorts only to a sample of the regular protocol usage between clients and servers and does not require any access to a protocol implementation or its source code, making it suitable to be used on open or closed protocols. Also, by taking advantage of some particular characteristics of network protocols, we devised more aggressive and optimistic approaches for both the language and the protocol state machine inference. Our experimental results have shown that ReverX is able to derive the language and state machine of the FTP protocol, using as little as 1000 messages of publicly available traces. Besides this experimental evaluation, we have successfully applied this approach with SMTP and POP protocols [30].

Currently, we are working towards extending the methodology to support binary-based protocol specifications with some promising results.

Acknowledgment

This work was partially supported by EC through project FP7-257475 (MASSIF) and by FCT through the LASIGE Multi-annual and CMU-Portugal Programmes and the project PTDC/EIA-EIA/100894/2008 (DIVERSE).

References

[1] V. Paxson, “Bro: A system for detecting network intruders in real-time,” in *Proc. of the USENIX Security Symposium*, 1998, pp. 3–3.

[2] A. Dahbura, K. Sabnani, and M. Uyar, “Formal methods for generating protocol conformance test sequences,” *Proceedings of the IEEE*, vol. 78, no. 8, pp. 1317–1326, 1990.

[3] J. Antunes, N. Neves, M. Correia, P. Verissimo, and R. Neves, “Vulnerability removal with attack injection,” *IEEE Trans. on Software Engineering*, vol. 36, pp. 357–370, 2010.

[4] J. Postel and J. Reynolds, “File transfer protocol,” RFC 959, 1985. [Online]. Available: http://www.ietf.org/rfc/rfc959.txt

[5] M. A. Beddoe, “Network protocol analysis using bioinformatics algorithms,” 2005, http://www.4tphi.net/~awalters/PI/PI.html.

[6] W. Cui, J. Kannan, and H. Wang, “Discoverer: automatic protocol reverse engineering from network traces,” in *Proc. of the USENIX Security Symposium*, 2007.

[7] C. De La Higuera, “Learning finite state machines,” *Finite-State Methods and Natural Language Processing*, pp. 1–10, 2010.

[8] N. Walkinshaw, K. Bogdanov, M. Holcombe, and S. Salahuddin, “Reverse engineering state machines by interactive grammar inference,” in *wcre*. IEEE Computer Society, 2007, pp. 209–218.

[9] J. E. Hopcroft, R. Motwani, and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley, 2006.

[10] R. Pang and V. Paxson, “A high-level programming environment for packet trace anonymization and transformation,” in *Proc. of the ACM SIGCOMM Conf. on Applications, Technologies, Architectures, and Protocols for Computer Communication*, 2003.

[11] C. Manning, P. Raghavan, and H. Schütze, “An introduction to information retrieval,” *Cambridge University Press*, 2008.

[12] C. de la Higuera, *Grammatical Inference: Learning Automata and Grammars*. Cambridge University Press, 2010.

[13] A. Biermann and J. Feldman, “On the synthesis of finite-state machines from samples of their behavior,” *IEEE Trans. on Computers*, vol. 21, no. 6, pp. 592–597, 1972.

[14] Y. Sakakibara, “Grammatical inference in bioinformatics,” *IEEE Trans. on Pattern Analysis and Machine Intelligence*, vol. 27, no. 7, pp. 1051–1062, 2005.

[15] D. Lo, L. Mariani, and M. Pezzè, “Automatic steering of behavioral model inference,” in *Proc. of the joint meeting of the European Software Engineering Conf. and the ACM SIGSOFT Int. Symposium on Foundations of Software Engineering*. ACM, 2009, pp. 345–354.

[16] L. Mariani and F. Pastore, “Automated identification of failure causes in system logs,” in *Proc. of the Int. Symposium on Software Reliability Engineering*, 2008, pp. 117–126.

[17] D. Lo and S. Khoo, “QUARK: Empirical assessment of automaton-based specification miners,” in *Working Conf. on Reverse Engineering*, 2006, pp. 51–60.

[18] V. Jacobson et al., “Tcpdump/libpcap,” http://www.tcpdump.org/, 1987.

[19] G. Combs et al., “Wireshark,” http://www.wireshark.org/, 2006.

[20] J. Rauch, “PDB: The protocol debugger,” in *BlackHat USA*, 2006.

[21] T. Beardsley, “Manual protocol reverse engineering,” BreakingPoint Systems, 2009.

[22] J. Caballero, H. Yin, Z. Liang, and D. Song, “Polyglot: Automatic extraction of protocol message format using dynamic binary analysis,” in *Proc. of the Conf. on Computer and Communications Security*, 2007.

[23] Z. Lin, X. Jiang, D. Xu, and X. Zhang, “Automatic protocol format reverse engineering through context-aware monitored execution,” in *Proc. of the Network and Distributed System Security Symposium*, 2008.

[24] W. Cui, M. Peinado, K. Chen, H. Wang, and L. Irun-Briz, “Tupni: Automatic reverse engineering of input formats,” in *Proc. of the Conf. on Computer and Communications Security*, 2008.

[25] G. Wondracek, P. Comparetti, C. Kruegel, E. Kirda, and S. Anna, “Automatic network protocol analysis,” in *Proc. of the Network and Distributed System Security Symp.*, 2008.

[26] G. Naumovich and N. Memon, “Preventing piracy, reverse engineering, and tampering,” *Computer*, 2003.

[27] P. M. Comparetti, G. Wondracek, C. Kruegel, and E. Kirda, “Prospex: Protocol specification extraction,” in *IEEE Security and Privacy*, 2009.

[28] M. Shevertalov and S. Mancoridis, “A reverse engineering tool for extracting protocols of networked applications,” in *Proc. of the Working Conf. on Reverse Engineering*, 2007.

[29] A. Trifilò, S. Burschka, and E. Biersack, “Traffic to protocol reverse engineering,” in *Proc. of the Int. Conf. on Computational Intelligence for Security and Defense Applications*, 2009.

[30] J. Antunes and N. F. Neves, “Diveinto: Supporting diversity in intrusion-tolerant systems,” in *Proc. of the Symposium on Reliable Distributed Systems*, 2011.

[8] Our experience with public FTP traces, for instance, shows that distinct message types appear with varying frequencies, making this solution inappropriate for these kind of traces.