ReverX: Reverse Engineering of Protocols

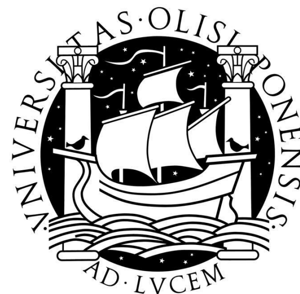
João Antunes, Nuno Ferreira Neves, Paulo Verissimo

DI–FCUL–TR–2011–01

DOI:10455/6699

(http://hdl.handle.net/10455/6699)

January 2011



Published at Docs.DI (http://docs.di.fc.ul.pt/), the repository of the
Department of Informatics of the University of Lisbon, Faculty of Sciences.

ReverX: Reverse Engineering of Protocols*

João Antunes Nuno Ferreira Neves Paulo Verissimo
LASIGE, 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 ReverX 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.

1 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 instance, 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 entire protocol space, to verify if a server is vulnerable to remote attacks [2]. The analysis and prevention of malware can also be significantly enhanced, for example, by discovering the protocol employed by a botnet to carry out command and control operations.

*This work was partially supported by the FCT through the LASIGE Multi-annual funding, the CMU-Portugal program, and by project PTDC/EIAEIA/100894/2008 (DIVERSE).

However, network components often rely on closed (or undocumented) protocols for their execution, sometimes to avoid integration with other third-party components by implementing some proprietary mechanism, other times trying to achieve security through obscurity. Occasionally, even if they are based on standard protocols, the developers add new features to differentiate their products from competing solutions, which cause reasonable changes to the underlying message formats and interactions. These extensions are normally undocumented or poorly described, rendering sizable parts of the specification to become undisclosed. In both these cases, closed or modified protocols, it is typically very hard to infer the specification because most of the reverse engineering work has to be carried out by hand, to make some sense of the seemingly arbitrary set of bytes that compose each message, and to determine its meaning and structure. Open protocols are usually well documented and their specification is readily available (e.g., all standard protocols from the IETF). However, even in this case, deriving a specification is difficult and time consuming because developers have to carefully analyze and translate the textual description of the protocol into some format readable by the tools.

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 technical report, we present a methodology for automatically inferring two automata that describe the protocol, where the first recognizes the formats of the messages (i.e., the language) and the second the order on which these messages can be exchanged (i.e., the state machine). This approach constructs the automata from the sequences of messages and protocol sessions that were observed in network traces, and then, generalizes and reduces the automata 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. However, we decided to focus on clear-text protocols, often used on network servers. By noticing that many of these protocols are text-based (e.g., HTTP, SIP, IMAP, FTP, Microsoft Messenger), we opted to narrow our approach on this kind of protocols and to take advantage of how text fields are usually organized and delimited in a message.

The methodology was implemented in ReverX, a tool that infers the protocol specification from a network trace containing a sample of protocol interactions. In a first phase, ReverX infers the protocol language (i.e., message formats) by constructing an automaton that recognizes and accepts messages present in the traces as well as different instances of the same message types. Then, a second phase extracts the individual protocol sessions, which together with the previously inferred message formats, are utilized to build an automaton that represents the protocol state machine.

A preliminary 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, FTP is a non-trivial protocol 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 [3], 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 and recall, even with training sets with a relatively small number of messages (around 1000 packets).

2 Related Work

In this technical report, 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 [4–6]. 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) [5, 7]. Some solutions also resort to specific rules or heuristics to aid the inference of the automata, such as heuristics to parse log files and to rewrite log events based on their attribute values, allowing similar log events to be identified and used to capture data flow information (e.g., sequences of related events) [8]. Although these techniques can produce useful models, their precision can be affected when dealing with larger and complex models [9]. Hence, some authors proposed to increase the precision of the automata by mining temporal properties from execution traces that capture relations between non consecutive and possibly distant events to steer the merging process [7].

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 [10–13]. 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) [14–17]. 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 [18], 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 [19]. Discoverer resorts to a different approach to derive more information about the messages [20]. 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 only 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 [21]. Message formats are inferred with equivalent techniques as [17]. 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 [22]. 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 cluster 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 is also 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 [23]. 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

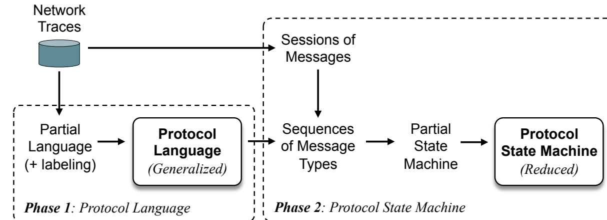


Figure 1: ReverX overview.

appear in the traces[1].

In this technical report, 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.

3 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 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 automata the *protocol state machine*. Together, the two FSM form the protocol specification that we intend to infer using our methodology.

One should note that, as in any other trace-based approach, the quality of the derived specification depends on the correctness and coverage of the protocol interactions present in the traces. Naturally, it will be impossible (unless we over-generalize) to infer a message type that is absent. Illegal or malformed messages will also affect the inference process by introducing incorrect formats or corrupting existing ones. To simplify the presentation, we will assume that protocol messages are not fragmented in several packets and that no encryption is performed. Since the parties involved in the communication can play distinct roles (e.g., client or server), and this may affect the allowed message types that

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

can be transmitted, we derive a separate specification for each role by looking at the direction of the traffic.

Figure 1 depicts an overview of the methodology implemented in the ReverX tool. The methodology is organized in two phases: In the first, ReverX iteratively constructs an automaton that accepts all messages of the network traces (*Partial Language*). Then, a generalization algorithm is employed to abstract parts of the FSM irrelevant to the protocol specification (*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 then converted into sequences of message types, to build an automaton that recognizes those sessions (*Partial State Machine*). The tool then identifies similar states to reduce the automaton (*Protocol State Machine*).

3.1 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.

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 all message fields ever 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.

3.1.1 Construction of the partial language automaton with frequency labels

A FSM is iteratively built such that it accepts every message in the network trace (Lines 16–30 in Algorithm 1). Message fields in text-based protocols are usually split by known delimiter characters, such as spaces or tabs. Therefore, by providing a regular expression with the field separators, it is possible to decompose a message (abstracted in Line 18). The fields and separators are treated as a sequence of symbols to construct the automaton. Whenever a field is rejected by the automaton, a new state and transition is added in order to

```
1  Function inferProtocolLanguage
2      Input: NetworkTraces : Messages of the protocol
3            T_1 : Minimum ratio of unique instances (0 ≤ T_1 ≤ 1)
4            T_2 : Minimum number of transitions (T_2 > 0)
5      Output: Automaton L ← (Q, Σ, δ, ω, q_0, F)
6
7      q_0 ← NewState()
8      Q ← {q_0}
9      Σ ← φ
10     δ(q, s) ← UNDEFINED for all domain
11     ω(q, s) ← 0 for all domain
12     F ← φ
13     L ← (Q, Σ, δ, ω, q_0, F)
14
15     // Iterative construction of the partial language FSM
16     foreach m ∈ NetworkTraces do
17         q ← q_0
18         foreach m_{1≤i≤|m|} do // for each message field
19             if δ(q, m_i) ≠ UNDEFINED then
20                 p ← δ(q, m_i)
21                 ω(q, m_i) ← ω(q, m_i) + 1 // inc. frequency
22                 q ← p
23             else
24                 p ← NewState()
25                 Q ← Q ∪ {p} // add new state to Q
26                 Σ ← Σ ∪ {m_i} // add symbol to alphabet
27                 δ(q, m_i) ← p // add transition
28                 ω(q, m_i) ← 1 // initial frequency label
29                 q ← p
30         F ← F ∪ {q} // add final state
31
32     // Generalizing the FSM
33     MinimizeFSM(L)
34     generalized ← TRUE
35     while generalized is TRUE do
36         Σ' ← {ANY}
37         δ'(q, s) ← UNDEFINED for all domain
38         ω'(q, s) ← 0 for all domain
39         LN ← (Q, Σ', δ', ω', q_0, F) // nondeterministic FSM
40         generalized ← FALSE
41         foreach q ∈ Q do
42             #trans_q ← |{δ(q, s) ≠ UNDEFINED , s ∈ Σ}|
43             freq_q ← Σ_{s∈Σ} ω(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 ∈ Σ : δ(q, s) ≠ UNDEFINED do
47                     δ'(q, ANY) ← δ'(q, ANY) ∪ {δ(q, s)}
48                     ω'(q, ANY) ← ω'(q, ANY) ∪ {ω(q, s)}
49                 generalized ← TRUE
50             else
51                 // transitions keep the same symbol
52                 foreach s ∈ Σ : δ(q, s) ≠ UNDEFINED do
53                     Σ' ← Σ' ∪ {s}
54                     δ'(q, s) ← {δ(q, s)}
55                     ω'(q, s) ← {ω(q, s)}
56         ConvertNFSM_to_DFSM(LN, L)
57         MinimizeFSM(L)
58     return L
```

Algorithm 1: Infer protocol message formats.

Session 1

```
IP1: USER clark
IP1: PASS kent
IP1: QUIT
```

Session 2

Session 3

```
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
```

Session 4

```
IP1: USER clark
IP1: PASS kent
IP1: CWD /home/clark
IP1: CDUP
IP1: QUIT
```

Session 5

```
IP2: USER bruce
IP2: PASS wayne
IP2: QUIT
```

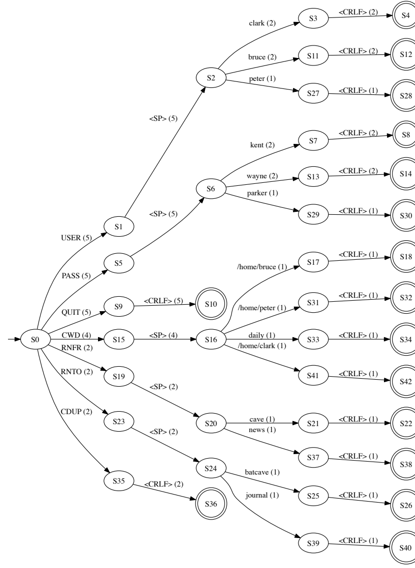
Figure 2: Example FTP network trace.

create a path accepting that message. In addition, all transitions are labeled with the number of times they were visited (Lines 21–28), to keep track of the frequency that each field has been observed in that particular place in the messages. The resulting frequency-labeled FSM is similar to a probabilistic automaton, where instead of a probability value, each transition has associated an absolute frequency.

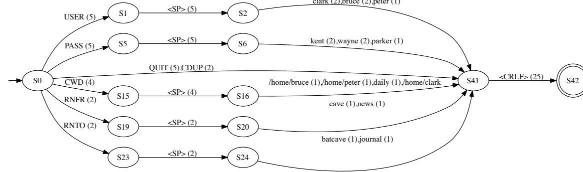
Figure 3 displays a simple example of inferring the protocol message formats from the network trace of Figure 2. The trace was obtained from five simple FTP sessions, just to elucidate the process of deriving a protocol specification using our methodology. To infer the protocol language recognized by the server, the trace was filtered so that it only contains 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 the RFC 959 [3]. Figure 3(a) shows the partial language FSM derived at the end of this phase, where “<SP>” and “<CRLF>” are the symbols of the field separator and message delimiter, respectively. Additionally, each transition is labeled with the frequency that it was visited (in parenthesis).

3.1.2 Generalization and Minimization

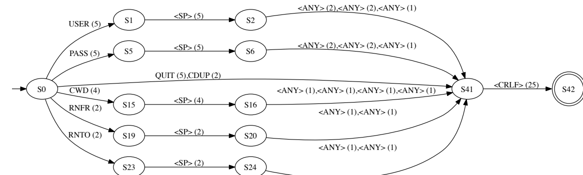
Once all messages have been processed and used to build the language FSM, the automaton is able to recognize the previously processed messages. In order to produce a more generic FSM that accepts other messages, one needs to identify and abstract the parts of the message format that are not fundamental to the specification (e.g., parameters of a command). At the same time, we also want to produce a concise automaton, with a reduced number of states and transitions.



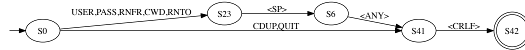
(a) Partial language automaton.



(b) Minimization.



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



(d) Generalization II: Determination and minimization.

Figure 3: Inference of the protocol language.

Otherwise, the same symbols could be scattered among equivalent states and transitions, augmenting the complexity of the derived specification.

We employ the Moore reduction procedure for deterministic finite automata minimization [24], which produces an equivalent FSM with the minimum number of states and transitions (Line 33). During the minimization process, equiv-

alent states are merged, i.e., states that accept the same symbols that lead to equivalent states. Merging two states also causes each equivalent transitions to be merged, and the resulting label is obtained by simply adding the frequencies together. Figure 3(b) shows the automaton calculated after minimization.

Once the automaton is minimized, we analyze the transition 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 commands with parameters and of responses. 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 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 the 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 is not bound to a limited number of fixed symbols. On 200 messages, the field could have 150 different values (e.g., if it corresponds to a pathname) 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 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 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 only needs to be greater than the maximum number of different command keywords that a protocol field can have, and hence it can be set to a generic value (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 makes the FSM become non-deterministic. Therefore, the algorithm resorts to a temporary (non-deterministic) automata LN that has the same states as the minimized automata L, but different transition and labeling functions (Lines 37–39). The LN automata is constructed as the states are evaluated for generalization (Lines 41–55). At the end of this stage, we employ a standard determination algorithm [24] to merge all non-deterministic 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 but 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 non-deterministic 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 non-deterministic automaton to a deterministic one, followed by the minimization operation is shown in Figure 3(d). This automaton also corresponds to the final FSM for the language, since the second loop iteration did not find any more states to generalize.

3.2 Inferring the State Machine

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

3.2.1 Obtaining the application sessions

The protocol state machine automaton is constructed from application *sessions*, each one corresponding to a sequence of messages exchange during the same interaction between parties. To identify individual sessions in the traces, we cluster messages that share similar network characteristics, such as source and

```
1  Function inferStateMachine
2      Input: Sessions sequences of message formats
3      Output: Automaton S ← (Q, Σ, δ, q0, F)
4
5      q0 ← NewState()
6      Q ← {q0}
7      Σ ← φ
8      δ(q, s) ← UNDEFINED for all domain
9      F ← φ
10     S ← (Q, Σ, δ, q0, F)
11
12     // Iterative construction of the partial state machine
13     foreach sx ∈ Sessions do
14         q ← q0
15         foreach m_{1≤i≤|sx|} ∈ sx do // for each message type
16             if δ(q, m_i) ≠ UNDEFINED then
17                 q ← δ(q, m_i)
18             else
19                 p ← NewState()
20                 Q ← Q ∪ {p} // add new state to Q
21                 Σ ← Σ ∪ {m_i} // add message type to alphabet
22                 δ(q, m_i) ← p // add transition
23                 q ← p
24         F ← F ∪ {q} // add final state
25
26     // merge states reached from similar message types
27     foreach q ∈ Q do
28         foreach p ∈ Q do
29             if ∃s ∈ Σ; r, t ∈ Q : δ(q, s) = r ∧ δ(p, s) = t then
30                 MergeStates(δ(q, s), δ(p, s))
31
32     // merge states without a causal relation that
33     // share at least one message type
34     reduce ← TRUE
35     while reduce is TRUE do
36         reduce ← FALSE
37         foreach q ∈ Q do
38             foreach p ∈ Q do
39                 // if there is not a casual relation
40                 if (∄s ∈ Σ : δ(q, s) = p ∨ δ(p, s) = q) or
41                    (∃s, t ∈ Σ : δ(q, s) = p ∧ δ(p, t) = q) then
42                     if ∃s ∈ Σ; r ∈ Q : δ(q, s) = r ∧
43                                        δ(p, s) = r then
44                         MergeStates(p, q)
45                         reduce ← TRUE
46         MinimizeFSM(S)
47     return S
```

Algorithm 2: Protocol state machine.

destination IP addresses and a temporal relationship. 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.

3.2.2 Construction of the partial state machine automaton

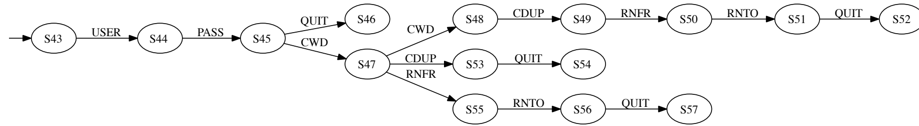
Analogous to the protocol language inference, we build an automaton that accepts the sequences of message types (the symbols of the alphabet) present in the application sessions (Lines 13–24). Since frequency information is not needed by Algorithm 2, the automata S does not define a labeling function. New states and transitions are added to the automaton whenever a distinct message type appears in the trace. In the end, the automata will 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.

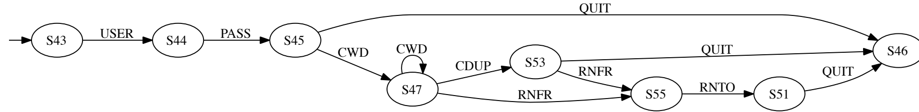
3.2.3 Reduction

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. Figure 4(b) shows the automaton after this procedure.

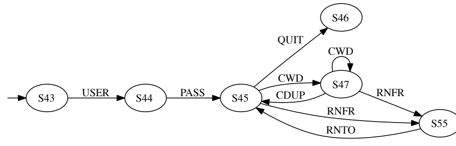
However, even some states that are reached from different message types, may be part of the same protocol state. For instance, after logging in, a user may create, edit, or delete files, all seemingly interchangeable protocol commands. With respect to the protocol state machine, the order of these messages is irrelevant after the user logs in, and they can be executed from a protocol state that accepts any of them. However, network traces are most probably incomplete, in the sense that many causal relations between protocol messages may be absent. To try to deduce a complete protocol state machine, in spite of the incompleteness of the network traces, the algorithm needs to make a few assumptions about the equivalence of some states. First, if there is a transition from one state $S1$ to state $S2$, but not vice versa, then they can never be



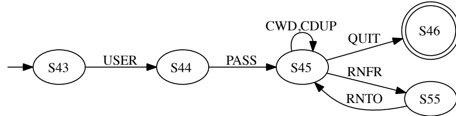
(a) Partial protocol state machine.



(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 state machine.

considered equivalent because this 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 sharing a similar transition (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 automata is the protocol state machine.

Figure 4(c) displays the automata 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)).

4 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. Additionally, since FTP is documented in an IETF RFC 959 [3], this facilitates the comparison between the inferred automata and the reference automata (manually produced from the textual description). The network traces were obtained from a public repository in the Web, to facilitate the reproducibility of the results, but more importantly, to preclude any bias to assist the reverse engineering task. In a sense, we want to determine how well the approach performs in a challenging scenario, where a user can collect the traces but cannot control or influence the clients and the servers. The section also analyzes the impact of different values of T_1 as the single generalization parameter (since T_2 is not so determinant) and the effect of the trace size on the created automata.

4.1 Experimental framework

4.1.1 FTP protocol

To assess the quality of our inference methodology, we reverse engineer a well known protocol. File Transfer Protocol (FTP) defines a standard way where clients can access files stored remotely in a server [3]. Clients have first to authenticate by providing a username and the respective password (USER command followed by the PASS command). An authenticated user can then navigate through the remote file system similarly to a local file system. The complete RFC 959 specification defines 33 commands, allowing clients to download or upload files, create directories, delete or rename files or directories, or even to obtain status information about files and directories.

4.1.2 Network traces

The evaluation uses publicly available FTP network traces[2]. Even though these traces had been previously anonymized [25], a few of the packets still contained malformed messages that had to be cleaned (such as, misspelled command names). Furthermore, we chose to concentrate on the specification of RFC 959 [3], 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).

4.1.3 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 FTP messages sent from clients are used in the experiments. Overall, 10

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

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. Second, each one of these automata is evaluated using 10 test sets. The test sets are generated by randomly choosing 4000 consecutive messages from the remaining trace file.

Two metrics assess the quality of the inferred automata. These metrics are inspired on widely used metrics from other fields, such as information retrieval [26, 27] and intrusion detection [28]:

- *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 probability that a randomly selected set of *valid* protocol messages (or protocol sessions) is accepted by the inferred automaton.

$$Recall_{FSM} = \frac{\#\text{ accepted messages (or sessions)}}{\#\text{ messages (or sessions)}}$$

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

$$Precision_{FSM} = \frac{\#\text{ accepted } \textit{valid} \text{ messages (or sessions)}}{\#\text{ accepted messages (or sessions)}}$$

Typically, it is very simple to have a recall of 1 (with a low precision) by having a very generic automata, which accepts all messages (or sessions). Therefore, the objective is to have a both a high recall and precision [27].

4.1.4 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.

[3] http://www.tcpdump.org/
[4] http://jnetpcap.com/
[5] http://www.graphviz.org/

4.2 Experimental results

All experiments described in this section set the value of T_2 to 30. Each point in the graphs corresponds to the average of 100 recall or precision calculations (10 experiments, each with 10 test sets).

4.2.1 Protocol language

To calculate the recall of the FSM of the language, we used the 4000 messages of each test set in the inferred automata to find out which packets were accepted or not. A recall with a value near 1 means that most of the messages are recognized, and that therefore, 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, but that could either be 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. 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.

Figure 5 shows the recall and precision of the protocol language automata inferred by ReverX. 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). By observing the graphs, it is possible to conclude that smaller training set sizes lead to worse quality automata. This is expected because if the trace is small, not enough message diversity is available to support the construction of a FSM that recognizes the whole language[7]. On the other hand, it is possible to see that 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 gen-

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

[7] In fact, the used trace suffered from this problem, since more than 70 percent of the messages are PORT requests.

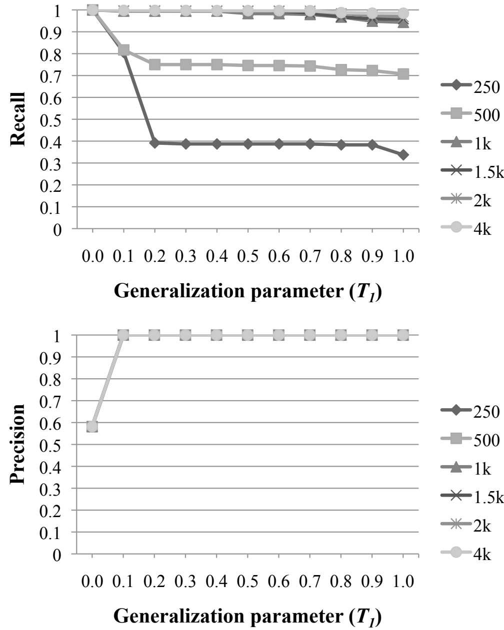


Figure 5: Language FSM evaluation.

eralize. 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 6 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. For instance, some parameters always take the same value due to the anonymization procedure (e.g., "SITE U6a1fb5bbU" or "USER anonymous"), and therefore, ReverX is unable to identify them as variable parameters. In practical scenarios, the impact of this limitation can be minimal because the missing parts of the inferred FSM correspond to the subset of the language that is not used by real client programs. If programs start to utilize these extra commands, then they should be inferred by ReverX because they will appear in the traces.

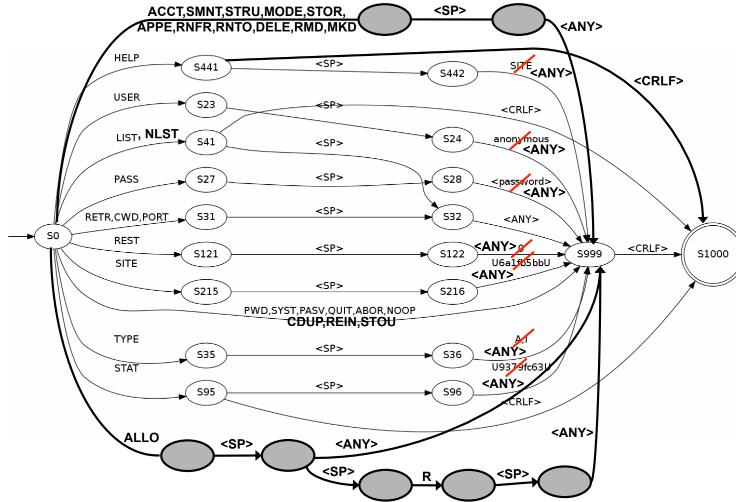


Figure 6: Inferred language FSM versus reference FSM for RFC 959.

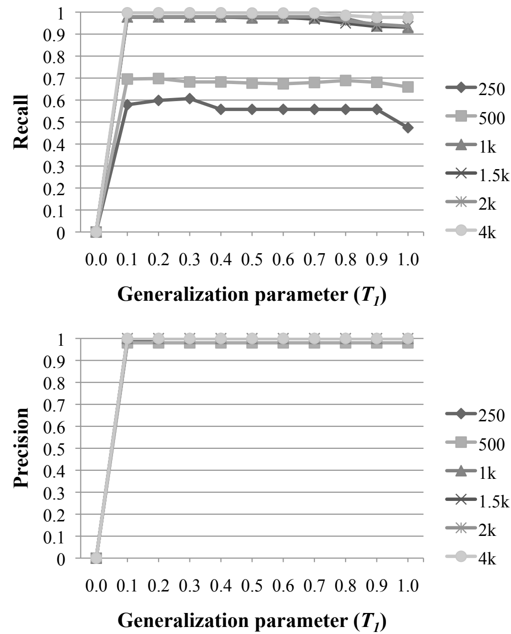


Figure 7: Protocol state machine evaluation.

4.2.2 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

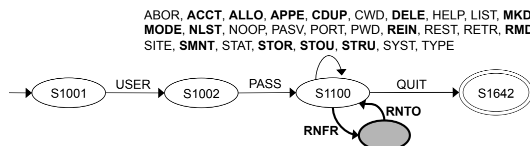


Figure 8: Inferred protocol state machine versus reference state machine from RFC 959.

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.

Figure 7 depicts the recall and precision graphs for the obtained protocol state machines. As with the protocol language inference, the best FSMs are built from larger samples of the training set (i.e., 1000 messages or more). Here, the impact of the size of the sample is more pronounced due to the fact that the training sets can have a small number of individual protocol sessions. In fact, we estimated that on average a session has 29 messages, and therefore, a set with 250 messages should 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 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 with only a single type of message. This causes an inferred state machine with a single state (i.e., a single message), which 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 or more) 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 8. 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. In fact, the message types in bold are also the message types missing from the inferred protocol language. 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

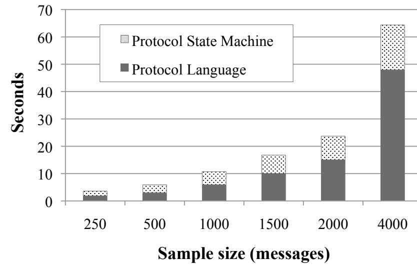


Figure 9: ReverX average execution time.

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 depicted in Figure 4.

4.2.3 Execution time

The average times that the tool takes to infer the protocol language and state machine are presented in Figure 9. ReverX is able to process 4000 messages and obtain the complete protocol specification in little over one minute (48 seconds to derive the protocol language and 16 seconds to infer the state machine). We also note that the time grows exponentially as the number of messages increases, which is due to the minimization steps of the methodology. However, recent advances in automata minimization algorithms could further reduce the execution time for larger sets. In any case, we were able to obtain good protocol specifications with as little as 1000 messages (and not much improvement is observed with larger traces) in a very short amount of time.

5 Conclusions

This technical report presents a new methodology and a tool to derive a protocol specification from the network traces. Our approach does not require any access to a protocol implementation or its source code, making it suitable to be used on closed protocols. The preliminary experimental results have shown that ReverX is able to derive the protocol language and state machine of the FTP protocol, using as little as 1000 messages of publicly available traces in about 10 seconds.

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] 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.

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

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

[5] 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.

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

[7] D. Lo, L. Mariani, and M. Pezzè, "Automatic steering of behavioral model inference," in *Proc. of the 7th joint meeting of the European Software Engineering Conf. and the ACM SIGSOFT Int. Symp. on Foundations of Software Engineering.* ACM, August 2009, pp. 345–354.

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

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

[10] V. Jacobson et al., "Tcpdump/libpcap," http://www.tcpdump.org/, 1987.

[11] G. Combs et al., "Wireshark," http://www.wireshark.org/, 2006.

[12] J. Rauch, "PDB: The protocol debugger," in *BlackHat USA*, 2006.

[13] T. Beardsley, "Manual protocol reverse engineering," BreakingPoint Systems, 2009.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] G. Naumovich and N. Memon, "Preventing piracy, reverse engineering, and tampering," *Computer*, 2003.

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

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

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

[22] 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.

[23] 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.

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

[25] 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.

[26] C. J. van Rijsbergen, *Information Retrieval.* Butterworth-Heinemann, 1979.

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

[28] D. Alessandri, "Attack-class-based analysis of intrusion detection systems," Ph.D. dissertation, University of Newcastle upon Tyne, UK, 2004.