Automatically Complementing Protocol Specifications From Network Traces

João Antunes and Nuno Neves
LASIGE, Departamento de Informática,
Faculdade de Ciências da Universidade de
Lisboa, Portugal
{jantunes,nuno}@di.fc.ul.pt

ABSTRACT
Network servers can be tested for correctness by resorting to a specification of the implemented protocol. However, producing a protocol specification can be a time consuming task. In addition, protocols are constantly evolving with new functionality and message formats that render the previously defined specifications incomplete or deprecated. This paper presents a methodology to automatically complement an existing specification with extensions to the protocol by analyzing the contents of the messages in network traces. The approach can be used on top of existing protocol reverse engineering techniques allowing it to be applied to both open and closed protocols. This approach also has the advantage of capturing unpublished or undocumented features automatically, thus obtaining a more complete and realistic specification of the implemented protocol. The proposed solution was evaluated with a prototype tool that was able to complement an IETF protocol (FTP) specification with several extensions extracted from traffic data collected in 320 public servers.

Categories and Subject Descriptors
C.2.2 [**Computer Systems Organization**]: Computer-Communication Networks—*Network Protocols*;
C.2.4 [**Computer Systems Organization**]: Computer-Communication Networks—*Performance of Systems*

1. INTRODUCTION
Network servers rely on protocols to offer services to their clients. Protocols prescribe how interconnected components should communicate by defining the rules and message formats that must be employed while exchanging data. As an example, the Internet Engineering Task Force (IETF) has been standardizing protocols for various applications, such as computer bootstrap in a networked environment [11], distributed name resolution [16] or remote email access [17].

EWDC '11, May 11-12, 2011, Pisa, Italy
Copyright © 2011 ACM 978-1-4503-0284-5/11/05... $10.00

Translating a human-readable specification (e.g., a RFC document) into a machine-readable format can be a cumbersome and error-prone task. Therefore, in the recent years, a few approaches have been devised to automatically infer an approximate protocol specification from network traces [7, 23, 2] or from the execution of existing implementations [5, 14, 9, 24]. These machine-readable specifications can then be employed in several areas, in particular in testing and security. For instance, the specifications can support the generation of test cases to evaluate if a particular server implements a protocol in a correct and secure way [1, 7]. Alternatively, they can be incorporated in filters of an application-level firewall, which rejects messages that violate the protocol [20] or they can be employed by intrusion detection systems to build signatures that are able to discover misbehaving components [18].

However, protocols are constantly evolving, as new functionality and different message formats are added, rendering the previously defined specifications incomplete or deprecated. Old specifications must therefore be updated with the new extensions, which typically requires a careful analysis to identify *where* the old specification was changed and *how* it should be updated. Sometimes developers must even incorporate multiple changes from more than one extension, making it an even more challenging task.

Currently, the existing solutions aimed at obtaining protocol specifications in an automated way do not make use of the older versions of the specification, creating the specifications completely from scratch. This means that to complement an existing specification, one must not only get data traces that cover the new features, but also re-create the old data traces in order to conserve the previous coverage of the protocol. Additionally, since these approaches ignore the old specification, one cannot easily identify the new part of the specification that pertains to the extensions, which might be useful, for instance, to prioritize the testing of the new features.

Our solution is based on protocol reverse engineering, but it takes advantage of the older version of the specification. Hence, the data traces it uses are only required to include information concerning the *new extensions* (although they can also have data pertaining to the old specification). At this moment, we are focusing on application-level clear-text protocols described by the IETF, widely used by many network servers, for example, FTP [19], IMAP [8], POP [17], or SMTP [12]. The methodology can be applied to both open and closed protocols[1]. In fact, closed protocols are a very interesting target for security purposes because, as opposed to open protocols, they are not subject to the public scrutiny and testing. Nevertheless, this approach can shed some light on the specification of closed protocols and on their latest changes. Even without a publicly available description, existing reverse engineering techniques can infer an approximate specification from network or execution traces, which can then be incrementally complemented using our methodology as newer traces are captured.

We implemented a prototype tool and evaluated our methodology with the current specification of the File Transfer Protocol (FTP, RFC 959 [19]) and with traffic data collected from 320 public FTP servers containing several extensions to the protocol. We found that the tool correctly complemented the specification with commands described in five different RFC extensions. The complemented specification also captured two non-standard protocol commands that were being used by a few FTP clients. This more complete specification is much closer to the real utilization of the protocol than the original document-based specification. It can provide valuable information as an unifying specification, which we intend to use in the future for testing and security purposes. Features not present in former versions of the specification should be given higher priority in testing. In particular, non-standard or undocumented extensions must be given special attention, since more obscure features are usually less tested.

2. METHODOLOGY

This section presents the methodology for complementing a protocol specification with new features or extensions. We focus on clear-text protocols, which are often used by network servers, such as many of the standard protocols published by the IETF. It is assumed that an older version of the specification already exists and that there are network traces with messages covering the new features (or that some implementation is available from which the network traces can be produced). Although in this paper we are using open protocols as an example, our solution can also be applied to closed protocols. The lack of a public protocol description would require an approximate specification to be inferred instead of being manually translated from the documentation, for instance, by reverse engineering the execution of a server [14, 9, 24] or the network traces [7, 23, 2].

In the solution, the original protocol specification is modeled as a finite-state machine (FSM) that describes the rules of communication between the clients and the servers. The automaton must capture both the language (i.e., the formats of the messages) and the state machine (i.e., the relation between the different types of messages) of the protocol. Separate specifications are devised for the client and server dialects, i.e., one FSM defines the messages recognized by the clients and their respective states, whereas the specification pertaining the server is defined by another.

[1] *Closed protocols* are protocols for which there is incomplete or no documentation to describe their behavior (e.g., message formats, states, transitions between states). *Open protocols* correspond to the opposite case, where this documentation is available.

```
Function extendSpecification
    Input: $A$: Automaton with the original specification of the protocol
           $NetworkTraces$: Messages of the protocol
           $T_1$: Minimum ratio of unique instances
           $T_2$: Minimum number of transitions
    Output: $A' \leftarrow$ Automaton with the extended specification of the protocol

    // Phase 1: Protocol Language
    $L \leftarrow$ empty automaton for the message formats
    $Formats \leftarrow$ list of message formats (regular expressions) taken from transitions of $A$
    foreach Format $f \in Formats$ do
        $Seq_f \leftarrow$ sequence of text tokens from $f$
        Add a new path to $L$ to accept $Seq_f$
    foreach Message $m \in NetworkTraces$ do
        $Seq_m \leftarrow$ sequence of text tokens from $m$
        If needed, add new path to $L$ to accept $Seq_m$
        Label newly created transitions with $New$
        Update frequency label of visited states

    $generalize \leftarrow$ True
    while $generalize =$ True do
        $generalize \leftarrow$ False
        foreach State $q \in L$
            if all transitions in $q$ are labeled as $New$ then
                $trans_q \leftarrow$ number of transitions defined in state $q$
                $freq_q \leftarrow$ frequency label of $q$
                if $trans_q / freq_q > T_1$ or $trans_q > T_2$
                    merge all transitions in state $q$
                    $generalize \leftarrow$ True
        Convert $L$ to deterministic automaton
        Minimize $L$

    // Phase 2: Protocol State Machine
    $A' \leftarrow$ automaton $A$ to be extended
    foreach Session $s \in NetworkTraces$ do
        $Seq_s \leftarrow$ Sequence of message formats from $L$ that accepts the sequence of messages of session $s$
        If needed, add new path to $A'$ to accept $Seq_s$

    foreach pair of States $\langle q_1, q_2 \rangle \in A'$ do
        merge states $q_1$ and $q_2$ if they are destination states of any two transitions in $A'$ with the same message format
    $reduce \leftarrow$ True
    while $reduce =$ True do
        $reduce \leftarrow$ False
        foreach pair of States $\langle q_1, q_2 \rangle \in A'$ do
            if there is a transition from $q_1 \rightarrow q_2$, but not $q_2 \rightarrow q_1$ then
                pair $\langle q_1, q_2 \rangle \leftarrow$ Non−Equivalent
            if there is no transition between $q_1$ and $q_2$ or no common transition defined in $q_1$ and $q_2$ then
                pair $\langle q_1, q_2 \rangle \leftarrow$ Non−Equivalent
            if pair $\langle q_1, q_2 \rangle \neq$ Non−Equivalent then
                merge states $q_1$ and $q_2$
                $reduce \leftarrow$ True
        Minimize $A'$

return $A'$
```

Algorithm 1: Methodology for complementing an existing specification from network traces.

Our approach consists in two distinct phases, one dedicated to the language of the protocol and another phase addressing its state machine. Algorithm 1 depicts the logical steps of the methodology to extend a given specification from network traces. Notice that the client and server specifications are treated separately, so the methodology has to be applied to both specifications. For this reason we use indiscriminately the terms specification, FSM, or automaton while referring to either the client or server specifications.

2.1 Phase 1: Protocol Language

One of the things that might change with a more recent version of a protocol is the set of messages that are accepted, i.e., the language it recognizes. Novel messages or formats might be introduced, and therefore, the first step consists in complementing the protocol language with the messages in the network traces.

First, we extract a list of the message formats that are already defined in the original specification (line 10, and also see Figure 1 for an example specification). Since we are addressing text-based protocols, message formats are modeled as regular expressions. For example, messages *USER jantunes* and *USER nneves* can be modeled as the regular expression *USER .**. The list of extracted message formats is a comprehensive account of the language recognized by the protocol, i.e., any protocol message must be accepted by at least one of the regular expressions, unless the message follows some extension yet to be specified.

We use the list of extracted message formats to build a FSM L for the original protocol language (lines 9–13). Each message format (regular expression) of the extracted list is tokenized in words and word separators (e.g., spaces, punctuation and any other special characters) (line 12). Hence, every message format corresponds to a sequence of tokens, and when added to L it will cause the creation of a new path of states and transitions (line 13). For example, a message *REST [0-9]+* would be divided in tokens *REST*, the space character, and *[0-9]+*, and the path would therefore be: state $S1$ is connected to $S2$ by transition *REST*, $S2$ is connected to state $S3$ by a transition accepting the space character, and finally $S3$ is connected to $S4$ by transition *[0-9]+*. At the end of this process, a FSM that can recognize all messages is produced, with the exception of the extensions.

The next step consists in identifying and adding new message formats not present in the original language of the protocol (lines 14–18). The network traces are parsed, and each message is tokenized into a sequence of words and word separators (line 15) and given to the automaton L. Whenever the automaton fails to recognize a new symbol (i.e., a word or a word separator) in a particular state, a new transition and destination state is created to accept it (line 16). The frequency that each state is visited during the construction of the new paths is recorded, and every new transition is labeled for later analysis (lines 17 and 18). This results in a FSM that accepts both the previously defined message formats and the new messages present in the network traces.

However, notice that the newly created paths are not generic enough to accept different instances of the same types of messages (e.g., if a path was created in L to accept the new message *SIZE xfig*, it would not accept similar requests with different parameters like *SIZE newfile*). Therefore, the new paths of states and transitions do not yet represent a message format, which must describe the composition and arrangement of fields of a given type of message. In our approach, a few additional steps must be followed in order to identify messages related to similar requests and to produce a regular expression that captures their common format. In another words, we must identify transitions in L that are associated with predefined values (e.g., command names), which should be explicitly defined in the new specification, and transitions concerning undefined data (e.g., parameters of commands).

To achieve this objective, we apply techniques similar to ReverX [2] where transitions with data that should be abstracted, such as specific parameters and other variable data, are identified and generalized (lines 20–31). Notice that only the transitions created for the new messages (in line 16) can be generalized and merged together. The other transitions correspond to the definition of message formats that were extracted from the original specification, and are consequently already generalized. Hence, we only analyze states in which all transitions are labeled as “New” (line 24).

Message fields associated with predefined values should appear often in the network traces (e.g., command *SIZE*), as opposed to the variable and less recurrent nature of the respective parameters (e.g., path names to several different files such as *xfig* or */libpcap.tar.Z*, just to name a few). Parameter data can therefore be recognized in states of the automaton that accept a wide range of different values (each one is a particular instance of that parameter field), and therefore, that have a large number of outgoing transitions. However, one can not rely solely on the individual frequency of each transition, or else commands that appear rarely in the traces could be misidentified as parameters. Therefore, we select states of the language FSM for generalization if at least one of these conditions are met (line 27):

- the ratio of the number of transitions leaving from a state over the total frequency of that state is above some threshold, T_1;
- the total number of transitions is larger than some predefined value, T_2.

Transitions of the selected states are then merged, i.e., a regular expression is produced to accept all values, and a new destination state is created by merging the former destination states of the transitions. After all states have been analyzed, the process is repeated if the FSM was modified by at least one generalization (lines 21 and 29). The resulting automaton thus recognizes the new language of the protocol, where each path, composed as a sequence of tokens that form a regular expression, corresponds to a different protocol message format.

2.2 Phase 2: Protocol State Machine

In the second phase of the methodology, we process individual application sessions from the network traces to complement the state machine of the protocol with the new message formats and corresponding protocol states.

Individual sessions are extracted from the traces in order to ascertain the logical sequence of types of messages that were exchanged between the clients and the servers (line 35). Different sessions can be distinguished by the client IP addresses and ports used in the connection, TCP sequence numbers, temporal gaps between messages, or simply by knowing which messages are used in the initial protocol setup as defined in the original specification.

Since the traces were already used to infer the protocol language, instead of the actual network messages, we use the respective message formats that were derived (i.e., the path in the automaton L that accepts the message). Thus, every application session, which is a sequence of messages, is converted into a sequence of message formats (line 36). Each sequence is fed to the FSM of the original specification and new states and transitions are added whenever the automaton fails to accept the complete session (line 37). For example, a session composed of messages *USER jantunes*, *PASS xyz*, and *REST 10* is first converted into the corresponding message formats *USER .**, *PASS .**, and *REST [0-9]+*; then, it is fed to the original specification, and all messages are accepted (see Figure 1). If the session included a novel message type such as *LPTR*, then a new transition would be created in the automation so that it could be accepted.

However, since we are dealing with potentially incomplete data sets (the network traces are a sample of the protocol utilization), the automaton only captures the sequence of messages exactly as they appear in the traces. Cycles and equivalent states must therefore be inferred. In this work, we use a similar technique to ReverX to identify and merge potentially equivalent states and cycles.

First, we identify states that are reached under similar conditions, i.e., from the same message format, because they probably represent the same protocol state. Hence, we merge any destination state of transitions that define the same message format (line 40). However, even some states that are reached from different message types may correspond to the same protocol state. For instance, after logging in, a user may create, edit, or delete files, all seemingly interchangeable protocol commands (i.e., the same protocol state with a cycle to itself). 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. To deduce a complete protocol state machine, in spite of the incompleteness of the network traces, we need to make a few assumptions about the equivalence of some states. First, if there is a transition from one state to another, but not vice versa, this establishes an explicit causal relation and thus they are deemed as *non-equivalent* (line 45-46). Second, protocol states without any explicit causal relation (i.e., without any transition between them or with transitions connecting the states in both directions) and with no common transitions (i.e., states accept completely different message formats), are also considered as *non-equivalent* (line 47–48). Consequently, any two states that were not labeled as non-equivalent are considered as *equivalent* and are therefore merged (lines 49–50). The automaton is then minimized (which will produce eventual cycles between interchangeable states and transitions) and this entire reduction procedure is repeated until no more states can be merged (lines 42 and 51). The resulting automata is the new complemented specification of the protocol state machine. The newly labeled transitions also reveal more clearly the changes brought by the network traces, which can help developers and testers to focus on the new part of the specification.

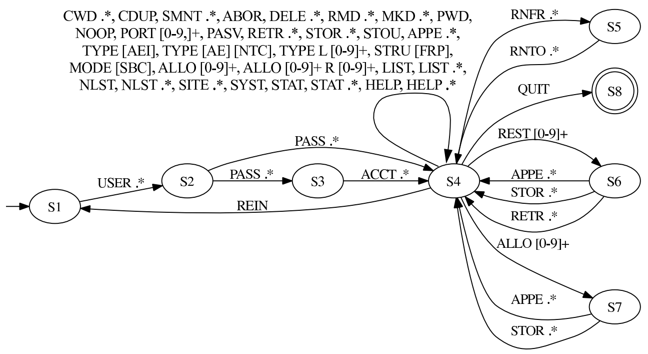
CWD .*, CDUP, SMNT .*, ABOR, DELE .*, RMD .*, MKD .*, PWD,
NOOP, PORT [0-9,]+, PASV, RETR .*, STOR .*, STOU, APPE .*,
TYPE [AEI], TYPE [AE] [NTC], TYPE L [0-9]+, STRU [FRP],
MODE [SBC], ALLO [0-9]+, ALLO [0-9]+ R [0-9]+, LIST, LIST .*,
NLST, NLST .*, SITE .*, SYST, STAT, STAT .*, HELP, HELP .*
RNFR .*
S5
RNTO .*
QUIT
S8
REST [0-9]+
PASS .*
USER .*
S2
PASS .*
S3
ACCT .*
S4
APPE .*
S6
STOR .*
S1
REIN
RETR .*
ALLO [0-9]+
APPE .*
S7
STOR .*

Figure 1: FSM for the FTP protocol (RFC 959).

3. EVALUATION

For the purpose of evaluation, we applied the methodology to complement a specification of a well-known protocol, with publicly available network traces that contained message types introduced in subsequent extensions. We chose the File Transfer Protocol (FTP) to illustrate the results because it is widely known and utilized. In addition, the FTP language and state machine are easily perceived from the examples, which makes it an interesting case study to show the potential results that can be obtained with the methodology. Since the server part of the specification is relatively simple—it mostly defines reply codes and implementation-specific response strings—, we opted to use and complement only the FTP specification related to the messages transmitted by the clients. Therefore, all automata and network traces concern the client-side of the protocol specification.

A client specification was manually produced for the original FTP protocol standard presented in RFC 959 [19]. Figure 1 shows the FSM for the original client FTP. It defines eight states, and the transitions are related to the various commands that can be executed in each state. For example, the first two states ($S1$ and $S2$) correspond to the initial authentication process where the client starts by indicating the username with command $USER$ and then provides the associated password with command $PASS$. The network traces were obtained from 320 public FTP servers located at the Lawrence Berkeley National Laboratory[2]. The traces span a period of ten days and contain over 3.2 million packets from 5832 clients.

A prototype tool was written in Java to implement the methodology. The tool uses as input the FSM of the original protocol specification and the FTP client requests (i.e., TCP messages from the traces transmitted to port 21). The tool follows the methodology as described in the previous

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

Table 1: Discovered message formats and respective RFC extensions.

Message Types	Introduced in
XCWD, XPWD	RFC 775
LPRT	RFC 1639
FEAT, OPTS	RFC 1839
EPSV, EPRT	RFC 2428
SIZE, MDTM, MLSD	RFC 3659
MACB, CLNT	non-standard
169 illegal requests	N/A

section. First, it produces a FSM recognizing the known language of the protocol, which is then extended with the new messages that were not recognized (*phase 1*). Then, the tool complements the protocol specification using the language inferred previously, placing the new message formats in the corresponding protocol states, as determined by the causal relations observed in the application sessions in the traces (*phase 2*).

Table 1 shows the new types of messages that the tool found in the FTP traces and the respective RFC document where they were published. A total of twelve new message types were extracted and their format inferred. Additionally, the tool detected 169 malformed protocol requests that consisted mainly of misspelled command names. To separate these erroneous messages from the rest, we just ignored command names that appeared only once in the traces, effectively preventing these messages from being further used in the experiments[3].

Among the twelve commands, the tool discovered two commands (MACB and CLNT) that were never published or documented by any RFC extension. MACB command is sometimes used by FTP clients running in the Macintosh Operating Systems (e.g., CuteFTP or WebTen) to transfer files in MacBinary mode, while CLNT refers to an obscure feature of a particular FTP client (NcFTP) apparently used to identify it and to access shell utilities. Little more information is available for these two non-standard commands, as they are not specified by any RFC or other official document. After identifying the new messages, the tool complemented the original specification with the observed extensions (Figure 2 shows the complemented specification with changes in bold). By analyzing the traces, the tool was able to discover the correct state of the protocol where the message formats were specified as extensions, i.e., the protocol state after the user logged in (state S4).

Naturally, the quality of the derived specification for the protocol language and state machine depends on the values of the generalization parameters (T_1 and T_2)[4] and on the comprehensiveness of the network traces, which should cover the protocol extensions one wishes to infer. Accordingly, any message type missing from the traces cannot be

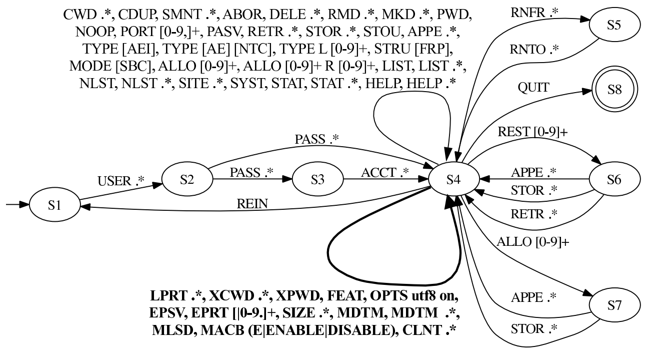
CWD .*, CDUP, SMNT .*, ABOR, DELE .*, RMD .*, MKD .*, PWD, NOOP, PORT [0-9,]+, PASV, RETR .*, STOR .*, STOU, APPE .*, TYPE [AEI], TYPE [AE] [NTC], TYPE L [0-9]+, STRU [FRP], MODE [SBC], ALLO [0-9]+, ALLO [0-9]+ R [0-9]+, LIST, LIST .*, NLST, NLST .*, SITE .*, SYST, STAT, STAT .*, HELP, HELP .*
RNFR .*
RNTO .*
QUIT
REST [0-9]+
APPE .*
STOR .*
RETR .*
ALLO [0-9]+
APPE .*
STOR .*
USER .*
PASS .*
PASS .*
ACCT .*
REIN
S1
S2
S3
S4
S5
S6
S7
S8
LPRT .*, XCWD .*, XPWD, FEAT, OPTS utf8 on, EPSV, EPRT [|0-9.]+, SIZE .*, MDTM, MDTM .*, MLSD, MACB (E|ENABLE|DISABLE), CLNT .*

Figure 2: FSM for the FTP protocol, complemented with message types and protocol states from subsequent extensions to the protocol (in darker).

extracted, and therefore cannot be used to complement the original specification. This problem can be addressed if one has access to a client and server implementation that supports the new features. In this case, the new functionality of the client can be exercised, thus producing a network trace that covers the entire protocol extensions, allowing the creation of a full protocol specification.

4. RELATED WORK

Our work aims at complementing existing specifications with new message formats and protocol states. To the best of our knowledge there is no work done with a focus on automatically complementing existing protocol specifications from network traces. There is, however, a substantial body of work dedicated to protocol specifications, such as in conformance testing or inferring automata.

Conformance testing emerged from the need to ensure the compliance of a given implementation with a predefined specification [13]. It usually resorts to finite-state machines to derive specific test sequences that traverse all transitions to verify the conformance of an implementation. Test sequences consist of sets of input and expected output obtained from the specification, with the purpose of checking if the input/output transitions are correctly executed by the implementation. Other approaches use passive testing to extract a set of invariants from the specification, and then check them against the traces produced by an implementation [6, 3, 25].

Automata inference is used to derive approximate protocol specifications when there is no formal specification available. The problem of inferring automata from incomplete data traces has been tackled in different research areas in the past, from natural languages to biology and to software component behavior [10, 4, 21]. 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) [4, 15].

A few works have also been focusing on the inference of protocol state machine specifications. Prospex employs taint analysis to obtain execution traces of a program for each session, which are then used to build an acceptor machine [7].

[3]Notice that any approach that uses data traces to infer or to learn some model must assume the correctness of its training data, so it is acceptable to ignore these erroneous messages from the evaluation.

[4]For a study about the impact of the generalization parameter values, T_1 and T_2, we refer the reader to the technical report [2].

PEXT utilizes network traces to infer an approximate state machine by clustering messages of the same type, based on a distance metric, and by analyzing the similarities between different sequences of types of messages present observed in the traces [22]. Trifilo et al. describes a protocol reverse engineering solution that resorts to the statistical analysis of network traces [23].

5. CONCLUSIONS

This paper presents a methodology to complement existing protocol specifications from network traces. Our solution has the advantage of not creating a complete specification from scratch, but by taking advantage of the previously defined (open protocols) or inferred (closed protocols) specifications and from network traces to capture new protocol interactions between the clients and the servers. The methodology was implemented in a prototype tool and was evaluated by complementing the standard FTP specification (RFC 959) with a trace collected from 320 public FTP servers. Several protocol extensions and two non-standard FTP types of requests were discovered and integrated in the FTP specification.

The proposed approach also has the advantage of obtaining a more complete and realistic specification because it integrates the rules and message formats from multiple and different extensions into a single specification. This unified specification captures the realistic utilization of the protocol, including unpublished or undocumented features present in the traces. In the future, we intend to extend this work to support the identification and subsequent removal of potentially obsolete parts of the specification, such as deprecated message types.

6. ACKNOWLEDGMENTS

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

7. REFERENCES

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

[2] J. Antunes, N. Neves, and P. Verissimo. ReverX: Reverse engineering of protocols. Technical Report TR-2011-01, Faculdade de Ciências da Universidade de Lisboa, Jan. 2011.

[3] E. Bayse, A. Cavalli, M. Núñez, and F. Zaïdi. A passive testing approach based on invariants: Application to the WAP. *Computer Networks*, 48(2):247 – 266, 2005.

[4] A. Biermann and J. Feldman. On the synthesis of finite-state machines from samples of their behavior. *IEEE Trans. on Computers*, 21(6):592–597, 1972.

[5] 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.

[6] A. Cavalli, C. Gervy, and S. Prokopenko. New approaches for passive testing using an extended finite state machine specification. *Information and Software Technology*, 45(12):837 – 852, 2003.

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

[8] M. Crispin. Internet Message Access Protocol – Version 4rev1 (IMAP). RFC 3501 (Proposed Standard), Mar. 2003.

[9] 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.

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

[11] R. Droms. Dynamic Host Configuration Protocol (DHCP). RFC 2131 (Draft Standard), Mar. 1997.

[12] J. Klensin. Simple Mail Transfer Protocol (SMTP). RFC 5321 (Draft Standard), 2008.

[13] R. Lai. A survey of communication protocol testing. *Journal of Systems and Software*, 62(1):21–46, 2002.

[14] 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.

[15] 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*, pages 345–354, 2009.

[16] P. Mockapetris. Domain names - implementation and specification. RFC 1035 (Standard), Nov. 1987.

[17] J. Myers and M. Rose. Post Office Protocol – Version 3 (POP). RFC 1939 (Standard), May 1996.

[18] V. Paxson. Bro intrusion detection system. `http://www.bro-ids.org/`, accessed in 2011.

[19] J. Postel and J. Reynolds. File transfer protocol (ftp). RFC 959, 1985.

[20] R. Russell. Iptables. `http://www.netfilter.org/`, first release in 1998.

[21] Y. Sakakibara. Grammatical inference in bioinformatics. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 27(7):1051–1062, 2005.

[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] 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.

[25] F. Zaidi, E. Bayse, and A. Cavalli. Network protocol interoperability testing based on contextual signatures and passive testing. In *Proc. of the ACM Symp. on Applied Computing*, 2009.