An Empirical Evaluation of a Language-Based Security Testing Technique

Muhammad AboElFotoh and Thomas Dean
Queen's University
{mha,dean}@cs.queensu.ca

Ryan Mayor
IBM Canada Ltd.
rmayor@ca.ibm.com

Abstract

Security testing of network applications is an essential task that must be carried out prior to the release of software to the market. Since factors such as time-to-market constraints limit the scope or depth of the testing, it is difficult to carry out exhaustive testing prior to the release of the software. As a consequence, flaws may remain undiscovered by the software vendor, which may be discovered by those of malicious intent. In this paper, we report the results of an empirical evaluation of applying a security testing approach and framework, previously tested in an academic setting, to the Distributed Relational Database Architecture (DRDA®) protocol as implemented by the IBM®DB2®Database for Linux®, Unix®, and Windows®product.

1 Introduction

The cheap availability of bandwidth has made global communication and collaboration easier, but the ease of interaction has also aided those with malicious intent. Thus the security of network applications is an increasingly important topic in both academia and industry. Conformance testing of these applications tends to focus on valid requests and obvious errors.

The protocols used by modern network applications have become languages in their own right, possessing syntax and semantics. We view the protocol as a programming language, and a data exchange between the client and server as a program. The system to be tested is viewed as an execution environment for the program. This opens up a variety of testing methods that may be used to evaluate the security of an implementation. Our approach tests the protocol implementation by violating the syntax rules and semantics of the protocol. In our approach, we capture packets, and after mutating the packets, inject them back into the network, in order to test the implementation's ability to handle erroneous cases. The mutations are based on the semantics of the protocol. Our previous work [5, 24, 29, 31, 34] has successfully applied this mutation approach to various protocols.

Some of the protocols that we have expressed using our approach (and tested implementations of) are X.509 [16, 14], OSPF [25] and SNMP [2], server message block (SMB) [21, 20, 9] and Apple file share (AFS) [15]. While all of these protocols are standard internet protocols, they are all relatively small protocols, and the servers that implement the protocols are similarly modest in size.

The goal of the work presented in this paper was twofold. The first was to evaluate the approach against a more complex protocol. The second was to extend the prototype [30] to automate the test planning phase of the approach.

In this paper we discuss our application

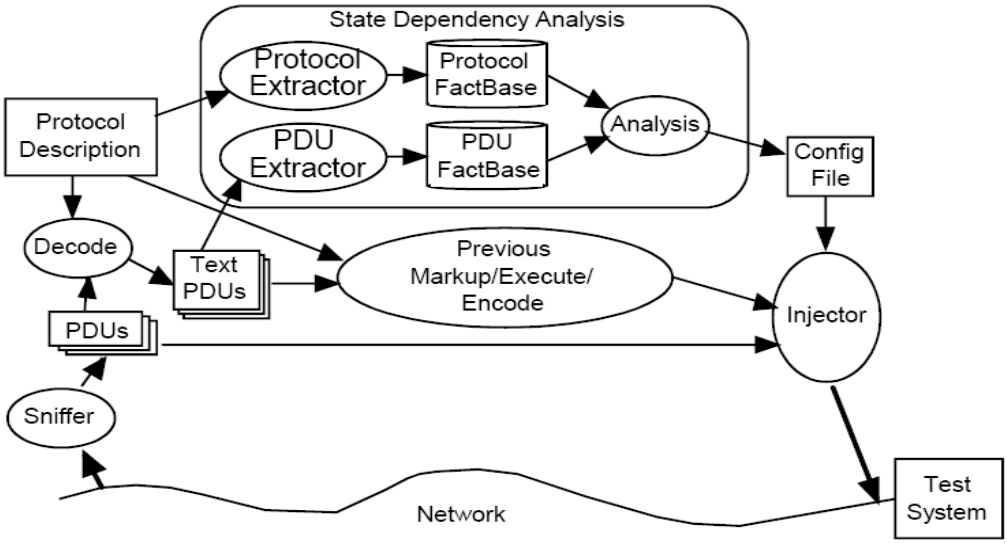

Figure 1: System architecture

of the approach to the distributed relational database architecture protocol (DRDA) [10, 11, 12]. This is a protocol originally developed by IBM which has been standardized by the Open Group. It is used as a middle level transport protocol between DB2 clients and servers. The protocol is a much more complex protocol, as it has over 200 rules and hundreds of element types, which supersedes the number of rules and element types for the protocols previously tested using this approach. This makes DB2, the implementation under test, a much more complex software application. New releases are also extensively tested before they are shipped by IBM.

The paper is composed of 7 sections. In Section 2 we give a short overview of our testing framework, as well as a short overview of the DRDA protocol. Section 3 discusses the experimental setup and the encoding of DRDA in our specification language. Section 4 discusses the extensions to the framework for DRDA. Section 5 presents the results of the DRDA tests. Section 6 discusses some related work, and the paper is concluded in Section 7.

2 Background

Figure 1 shows the system architecture of our prototype protocol tester. It is implemented in a mixture of Java®, C, TXL [4] and grok [13]. A network listener is used to capture the packets that form the messages (Protocol Data Units or PDUs) that are exchanged by the client and server (lower left). This is done while some set of tests is run such as conformance tests, or regression tests or some other test suite. The protocol is specified in the SCL language [24], shown in the upper left. The SCL language uses XML markup to add constraints to the Abstract Syntax Notation dot One (ASN.1) [6] protocol description standard. The protocol description is used to decode the messages into a textual form. The protocol description is also transformed into a fact base representation giving both the structure and the constraints in TA form [27]. A set of facts representing the captured PDUs is also extracted. As part of the extraction process, each field in the captured PDUs is given a unique name which serves as a link between the fields and the facts. Both the PDUs and the protocol description are used by a mutation

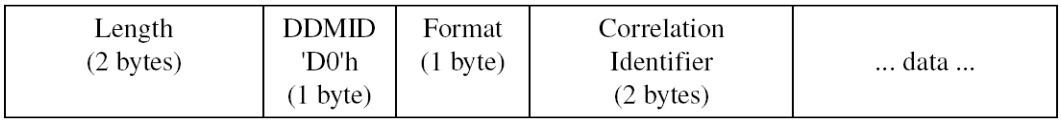

Figure 2: A single Data Stream Structure (DSS)

engine which checks to see which constraints in the protocol description are present in the captured PDUs and uses that information to generate variants of the PDUs that violate the constraints. In the existing prototype, while the low level mutation process is automated, the planning phase based on the protocol description was still largely manual. The modified PDUs are then sent to the test system to see if they cause any faults in the system. If the system engages in unexpected behaviour as a result of the fault injection, such as unexpectedly shutting down or crashing, then the modified PDUs can be used by an attacker to perform a denial-of-service (DoS) attack, or, in some cases, compromise the system. If the mutated message is not the first message in the sequence, the previous messages must be retransmitted to place the test system into the same state. The fact bases extracted from the PDUs are analyzed as part of state dependency analysis to create a script which provides the injector with information about which fields of the messages must be modified to maintain any state dependencies of the protocol.

DRDA describes the contents of all the commands, data objects and replies that flow between the client and the server. DRDA is built on the Distributed Data Management (DDM) architecture, which it uses as the transfer syntax for the exchange of DRDA messages. DRDA messages are carried in one of three data stream structure (DSS) messages, namely Request DSS, Object DSS or Reply DSS. A DRDA PDU can consist of a single DSS, or a sequence of DSS messages. More than one message can be pipelined into a single PDU, and more that one DSS might be used by a single command or reply. The format of a single DSS is shown in Figure 2.

The data carried inside these structures can be one or more commands, objects or replies to commands. The format of a single command/ object/reply is shown in figure 3. The length fields in the structures shown in figure 2 and 3 specify the total length of the structure, including the 2-byte length field. In a DSS, the length field is followed by a 1-byte DDM identifier ('D0' h). The identifier is then followed by a 1-byte integer field, the format field, which specifies whether the message is a command or an object or a reply. Following the format field, is the correlation identifier 2-byte integer field which is used to identify structures which belong to particular commands and replies. The code point field in Figure 3 is used to specify the type of a message. DRDA has 580 code points, some of which are specific to the communication protocol, and some of those structures are of a primitive type, that can be directly expressed using SCL's primitive types.

3 Experimental Setup

The system under test was DB2 Open Beta Viper version 9.5. The software was installed on a dualcore machine with 1 GB of RAM, with an instance of DB2 acting as the server and listening on TCP/IP socket port 50000. The DB2 client, also on the same machine, using another instance of DB2, connects to the server via the loopback interface. The server's authentication method was kept on the default settings. Some of the test sequences were obtained from the samples included in the evaluation version of DB2 version 9.5. Others were generated using custom queries to target specific DRDA messages. The initial state of the database for each sequence was backed up, and the injector runs a shell script after each individual sequence of PDU is retransmitted, restoring the database to its original state.

Length (2 bytes)	Codepoint (2 bytes)	... data ...

Figure 3: Command/Object/Reply structure

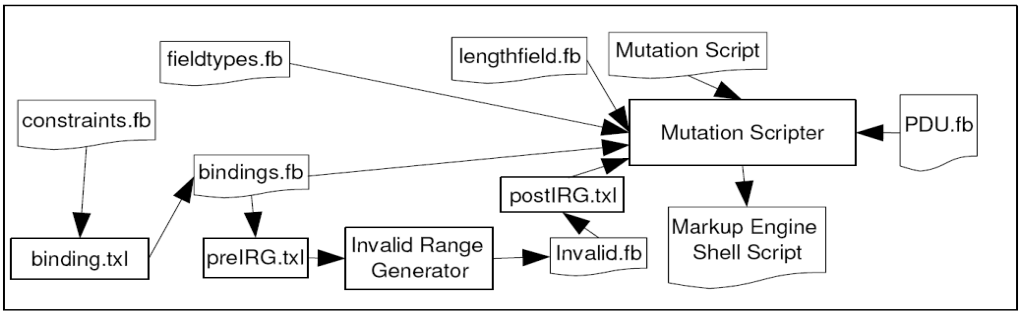

Figure 4: Test planning architecture

4 Framework Extensions

Several omissions in the existing implementation of our testing framework were uncovered during this research. Several of these were elements of the framework that had been designed, but not yet implemented, while others were new issues. This section examines some of the extensions made as part of the experiment.

4.1 Test planning

The link between the protocol description and the mutator was still manual in the prototype. As part of this project, a prototype test planner was completed. This was the most significant extension to the framework as part of the project. The low level markup and execution engine of the mutator uses a shell script that passes a list of the unique names of the fields that are to be mutated, and the specific mutations that are to be performed. Figure 4 shows the new architecture of the Test planner. It consists of five steps: pre-binding, binding, planning, invalid range generation and script generation.

4.1.1 Pre-Binding

The pre-binding is part of the PDU fact extraction process. It generates unique names for each of the fields, and extracts from the protocol fact base the information about any constraints that apply to fields that are in the particular sequence of PDUs that are being mutated.

4.1.2 Binding

The binding phase takes the information that was assembled by the pre-binding step matches the unique names and constraints. It generates a simple set of variables which are substituted into the constraints and unified with the unique names. The constraints are converted into simple arithmetic/ logical expressions and classified as one of three types, length, value or cardinality. Figure 5 shows a subset of the output of the binding phase for one of the tests. In the figure, there are three constraints (lines starting with Expr). Two are length constraints, the other is a value constraint. The other lines, starting with BoundTo, bind unique names of PDU fields to the variables in the expressions.

```
BoundTo Rqsdss_cmd x4
BoundTo
Rqsdss_header_DssHeader_lenField
x5
Expr "x4 = ((x5) - 6)" LENGTH
BoundTo
RDBCMM_llcp_LLCp_codePoint x6
Expr "x6 == 8206" VALUE
BoundTo RDBCMM_rdbnam x7
BoundTo
RDBCMM_llcp_LLCp_lenField x8
Expr "x7 = ((x8) - 4)" LENGTH
```

Figure 5: Sample output of binding

```
setInvalidValue all

setValue "MAXVAL" where type like INT

setValue "0" where type like OCTET_STRING

setValue "MAXVAL" where type like OCTET_STRING

remove all

permute "0" where type like SEQUENCE

permute "0" where type like SET
```

Figure 6: Default Mutation Script

4.1.3 Invalid Range

The output of the binding stage is fed to a TXL script which prepares the value constraint equations for parsing by the Invalid Range Generator. The Invalid Range generator is a C++ program that parses the expressions and inverts the range expressions. For example, expression x1>5 and x1!=7 becomes x1<=5 or x1==7. The output of the invalid range generator is a fact base which contains a table of the fields' variables and a corresponding value which violates the range expression. As suggested by Beizer [1], the default behaviour of the framework is to generate cases with single mutations. For multiple mutations per test case, the framework user has to instruct the framework to generate multiple mutations per test case. The result of these two steps is a sequence of mutations to be performed. Figure 6 shows a sample of the output of these two steps. All three of these instructions are to set particular fields to particular values.

4.2 Mutation Script Generation

This stage, the final stage in the test planning process, takes as input the binding fact base from the binding stage; the field types fact base, which contains all the fields in the protocol description, with their corresponding types; the length field fact base, which contains all the affecter and affected fields for each length constraint in the protocol description; the output from the Invalid Range Generation phase, as well a script which describes the overall mutation strategy. This script can be modified by the framework user for customized mutation strategies. Figure 7 shows a single mutation shell script command.

4.3 Markup Engine

In the previous versions of the framework, there was no handling of length fields as it was not required. However, the DRDA protocol messages do have a length field which holds the message length. If the creation of a test case involved an insertion or deletion of a DRDA child element carried by a DRDA message, then the length field value of the parent DRDA messages have to be modified to account for the insertion/deletion of that child element. Our goal was to generate a single insert/delete field mutation. Leaving any parent structures' field values' unfixed would mistakenly result in two mutations, a length field value mutation as well as the insert/delete field mutation.

Figure 8 shows an example of the length fields affected by an insert/delete field mutation. The DRDA PDU shown contains two DSSs, the mutation is in the second DSS in a child message structure. The numbers proceeding the element label indicate the "level" of the element. For example, Affected L.F. (length field) 2 is a top-level length field for DSS 2. Affected L.F. 2.1 is an affected length field of a child of DSS 2, and Affected L.F. 2.1.1 is an affected length field of a grandchild of DSS 2.

```
"txl" "$1" "$2" "-o" "$3.tc3" "-" "--uniquefieldid"
"sendPK13_PDU_seqOfDss_seqOfDss_PDU_1_seqOfDss_Rqsdss_header_DssHeader_
constant" "--error" "ErrASN" "setASNValue" "MAXVAL"
```

Figure 7: Shell script line from Mutation Scripter

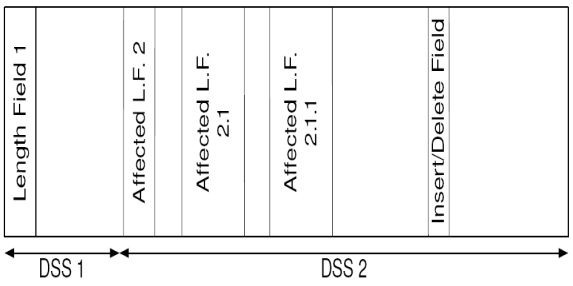

Figure 8: Insert/Delete mutation example

4.4 Constraints

Since most of the protocols previously handled did not have very complex constraints, the constraint language was also somewhat simplistic. As a result, it was not able to handle all of the constraints in the DRDA protocol. In particular, values of fields were limited to an enumeration of values, or range of values as shown in Figure 8. Constraints with multiple ranges were not supported, nor were multiple fields supported in the same value constraint. Figure 9 shows an example of one of the constraints for the Description Error Code (DSCERRCD) structure in DRDA. The constraint describes three ranges of values (1..7, 11..13, 32..26) as well as the individual values 21, 22, 41, and 42. The syntax length constraints also had to be extended to include conditions and multiple fields. In particular there is one structure type where an array of fields has a default length if one field has the value 66, or has a length given by another field if the first field has a value other than 66. Optional elements, while available in ASN.1, were not fully implemented in the existing prototype and had to be added.

4.5 Decoder

In previous protocols, all PDUs fit into single packets simplifying the decoder. Several of the PDUs in the DRDA packets spanned multiple packets. The network listener generates a single dump file containing all of the packets. The new decoder creates a textual representation of the network traffic captured, a Packet Detail Markup Language (PDML) file from the binary dump file. The PDML file is then fed into a utility program, called the Payload Extractor, which extracts the payload from the PDML file, and constructs the packet sequence, one binary file per packet. A shell script uses the PacketDecoder, a java program which can decode single binary PDUs using the protocol description, to decode the first packet file, if successful then the script moves on to the next packet. However, if the packet cannot be decoded then the packet is concatenated with the next packet in the sequence, and then decoded. If successful, then these two consecutive packets are stored as one binary PDU in one file. If the packet which is yet to be decoded has been concatenated with all the packets proceeding this packet in the sequence, and still cannot be decoded, then the framework throws an error. Each PDU file is appropriately named according to the PDUs position in the sequence. A list of the packet assembly is generated for later use by the injector. The decoder then translates each PDU to its textual form as had been done for previous protocols. While the implementations of the protocols previously tested uses sets of items, they did not use the SET construct for records. This construct describes a heterogeneous list of fields, but allows the order to change. DRDA uses this construct for some of the message classes, and thus the decoder was extended to support this construct.

4.6 Injector

Since the tests involved commands that produced side effects in the server, a means to restore the server to the initial state was needed. The injector was extended to run an external command to return the database to the known initial state. The external command used to restore the database to a known state was an invocation of a shell script which restored a backed up version of the DB2 instance's 'db2instX' directory and the DB2 instance's 'sqllib' directory.

Since some PDUs span more than one network packet, the injector must separate mutated request PDUs before retransmission and reassemble response PDUs as they are received. To maintain the protocol independence of the injector, the decoder provides a list that can be used for this purpose. The injector was also refactored to separate the state dependency rewrites from the injection.

5 Test Results

A total of 26 test sequences were run. The shortest test sequence contained 6 PDUs, while the longest sequence was 127 PDUs. The average sequence length was 33 PDUs. The minimum number of mutant PDUs for a sequence was 121 mutations, The maximum number of mutant PDUs was 6874 and the average number of mutants for a sequence was 1654.

The average time to generate a set of test cases was 8 minutes, and the average total test time was 83 minutes. Details of the test results are shown in Table 1.

As of DRDA version 3, there are 21 DDM Command Objects and 22 DDM Reply Objects and Messages used by DRDA [12]. Of the 21 DDM Command Objects, 18 were described. The remaining three DDM Command Objects SYNCCTL, SYNCRSY and DRPPKG were not described as they were not encountered throughout the testing phase. Of the 22 DDM Reply Objects and Messages, 19 were described. The Sync point control Reply Data (SYNCCRD), the Sync point Log (SYNCLOG), and the Sync point Resynchronization Reply Data (SYNCRRD) codepoints (message types) were not described as they were not encountered throughout the testing.

Two faults in the server were found by our tests. One was a previously reported fault, the other was a new fault. The new fault was a security fault that crashed the DB2 server instance, resulting in a Denial-of-Service (DoS). The fault has been remedied and has been released as part of a scheduled service patch for the affected version of the server.

Seq#	Length (# of PDUs)	# of mutant PDUs	Generation time (mins)	Run time (mins)
1	19	1018	6	72
2	8	6874	39	275
3	7	268	1	13
4	7	889	5	42
5	7	121	1	6
6	14	1266	6	56
7	7	121	1	6
8	11	247	1	11
9	44	4413	23	207
10	11	724	4	32
11	8	700	4	28
12	26	1056	6	90
13	7	764	4	36
14	68	1836	9	190
15	26	1368	8	117
16	40	696	4	33
17	64	1200	6	124
18	125	3052	11	71
19	127	3924	14	91
20	44	1322	7	62
21	35	1706	9	80
22	115	3211	11	74
23	11	706	4	26
24	6	738	4	31
25	14	632	3	28
26	25	4155	24	354

Table 1: Test Results

6 Related Work

Other methods have also been used to test state based protocols. One approach is to model the protocol using automata based approaches [22, 23]. However this is a heavy weight solution in that the interpreted state machine is essentially a fully functional client. Approaches mixing abstract specifications such as grammars and code are also possible [19, 18, 28, 32]. However what is common to all of these approaches is that the state space of the server (or client)

must be specified in some way.

The PROTOS project [18] at Oulu University uses a protocol grammar using higher order attribute grammars to generate variant PDUs. The grammar specifies the possible PDUs right down to the values of fields. The grammar is modified using a script to allow the desired errors and then a walker walks the grammar tree, automatically generating the PDUs and analyzing responses. The higher ordered attribute grammar actions are custom written Java routines. They are triggered when the walker reaches specific grammar nodes. Examples of actions include copying values from result PDUs to client PDUs (including arbitrary computation such as password encryption) and computing checksums of PDUs.

The PROTOS approach was extended at Cisco [33, 32] This approach uses a user-defined external callback routine. This routine is custom written for each protocol. It is used to verify the return PDUs and make corresponding changes to the test PDUs.

Recent work [17] by Jing et al. has applied an ISO testing language, Testing and Test Control Notation version 3 (TTCN-3) to mutation testing, specifically against the OSPF protocol. TTCN-3 is a test specification language that includes the ability to model the state transitions that the implementation may undergo and any changes that must be made to the test data. Thus it is capable of modeling our approach. In addition, they model the verification by adding a forced reset to a known initial state. This would be similar to sending a DRDA interrupt command. The advantage of our approach is that we need not describe the entire state model, only the trace dependencies of the protocol.

While the general technique is similar, our approach is different. We capture a valid set of data by sniffing the network and transforming it to generate alternate PDUs. We also are working on automatically generating the test plans based on the syntax and semantics of the protocol without manual intervention.

The mutation technique can be thought of as a variation of Syntax Testing [1], Model-Based Testing [7, 8] and Random Testing (Fuzzing) [3, 26]. In the Syntax Testing approach, syntax and semantic errors are intentionally made to produce unexpected variations in the protocol's messages to attempt to expose vulnerabilities. Syntax Testing can be considered a subset of Model- Based Testing. Other recognized areas of model based testing include test generation by theorem proving, constraint logic programming, model checking and symbolic execution.

We share the mutation of input test data with the random testing approach and many of the techniques in our syllabus of testing strategies are based on random testing. However, we use the linguistic model to plan the test strategy and to better determine where random testing is best applied, and we do not limit the applicable test strategies to randomizing selected fields.

7 Conclusions

We have shown that our approach can scale to handle protocols such as DRDA. A single student was able to describe 200 out of 580 message element types over eight months while at the same time extending the testing framework. The description and extended framework enabled us to run a total of 26 tests, generating a total of 27925 mutants. Two software defects were found, one of which was new. The few defects is not surprising given the extent to which new versions of DB2 are tested before they are released.

Trademarks

References

[1] B. Beizer. *Software Testing Techniques, 2nd Edition.* Van Nostraad Reinhold, New York, 1990.

[2] J. Case, M. Fedor, M. Schoffstall, and J. Davin. Simple Network Management Protocol. *Internet RFC 1157*, 1990.

[3] T.Y. Chen and F.C. Kuo. Is Adaptive Random Testing Really Better than Random Testing. In *Proc 1st Int. Workshop on Random Testing*, pages 64–69, Portland, USA, July 2006.

[4] J. Cordy. The TXL Source Transformation Language. *Science of Computer Programming.*, 61(3):190–210, August 2006.

[5] T.R. Dean and G.S.N Knight. Applying Software Transformation Techniques to Security Testing. *International Workshop on Software Evolution and Transformation, Delft, Netherlands*, pages 4952–4952, November 2004.

[6] O. Dubuisson. *ASN.1 Communication between Heterogeneous Systems.* Academic Press, San Diego, 2001.

[7] I.K. El-Far and J. Whittaker. *Model-based Software Testing.* Encyclopedia on Software Engineering, ed. J.J. Marciniak, Wiley, 2001.

[8] E. Farchi, A. Hartman, and S.S. Pinter. Using a Model-Based Test Generator to Test for Standard Conformance. *IBM Systems Journal*, 41(1):89–110, 2002.

[9] The Open Group. *Protocols for X/Open PC Interworking: SMB, Version 2, ISBN 1872630-45-6.* The Open Group, October 1992.

[10] The Open Group. *DRDA, Version 3, Volume 1: Distributed Relational Database Architecture (DRDA).* The Open Group, January 2004.

[11] The Open Group. *DRDA, Version 3, Volume 2: Formatted Data Object Content Architecture(FD:OCA).* The Open Group, January 2004.

[12] The Open Group. *DRDA, Version 3, Volume 3: Distributed Data Management (DDM) Architecture.* The Open Group, January 2004.

[13] R. Holt. Introduction to the Grok Language, http://plg.uwaterloo.ca/∼holt/papers/grok-intro.doc, last accessed, May 2008.

[14] IETF. Public-Key Infrastructure (X.509), http://www.ietf.org/html.charters/pkix-charter.html, 2004., last accessed Aug 10, 2006.

[15] Apple Computer Inc. Apple Filing Protocol Programming Guide Version 3.2.

[16] RSA Data Security Inc. PKCS#7-Cryptographic Message Syntax Standard, 2004.

[17] C Jing, Z Wang, X Shi, X Yin, and J Wu. Mutation Testing of Protocol Messages Based on Extended TTCN-3. In *Proc 22nd International Conference on Advanced Information Networking and Applications*, 2008.

[18] R. Kaksonen. *A Functional Method for Assessing Protocol Implementation Security (Licentiate thesis).* Espoo. Technical Research Centre of Finland, VTT Publications 447. ISBN 951-38-5873-1, 2001.

[19] R. Kaksonen, M. Laasko, and A. Takanen. Vulnerability Analysis of Software through Syntax Testing, http://www.ee.oulu.fi/research/ouspg/protos/analysis/WP2000-robustness/index.html, 2001., last accessed May 2008.

[20] P. Leach and D. Naik. *Draft-leach-cifs-v1-spec02: A Common Internet File System (CIFS/1.0) Protocol, Internet Draft. IETF, March 13, 1997.* IETF, 1997.

[21] P. Leach and D. Perry. CIFS: A Common Internet File System, Microsoft Internet Developer, MicroSoft Developer Network, http://www.microsoft.com/mind/1196/cifs.asp.

[22] D. Lee, K. Sabnani, D. Kristol, and S. Paul. Conformance Testing of Protocols Specified as Communicating Finite State Machines - A Guided Random Walk Based Approach. *IEEE Transactions on Communications*, 44(5):631–640, 1996.

[23] D. Lee and M. Yannakakis. Principles and Methods of Testing Finite State Machines - A Survey. In *Proceedings of The IEEE*, volume 84, pages 1090–1123, 1996.

[24] S Marquis, T. Dean, and G.S.N Knight. SCL: A Language for Security Testing of Network Applications. In *Proc. CASCON 2005*, pages 155–164, Toronto, Canada, Oct. 2005.

[25] J. Moy. OSPF Version 2. *Internet RFC 2328*, 1998.

[26] D. Owen, D. Desovski, and B. Cukic. Random testing of formal software models and induced coverage. In *Proc 1st Int. Workshop on Random Testing*, pages 20–27, Portland, USA, July 2006.

[27] Holt R. TA: The Tuple Attribute Language, Department of Computer Science, University of Waterloo, July 2002 http://plg.uwaterloo.ca/∼holt/papers/ta-intro.htm, last accessed May 2008., 2002.

[28] Sourcefire. SNORT web site at http://www.snort.org, last accessed Aug 10, 2006.

[29] O. Tal, S. Knight, and T.R. Dean. Syntax-based Vulnerability Testing of Frame-based Network Protocols. In *Proc. 2nd Annual Conference on Privacy, Security and Trust*, Fredericton, Canada, October 2004.

[30] Y. Turcotte, O. Tal, S. Knight, and T. Dean. Universal methodology and tools for syntax-based vulnerability testing of protocol implementations. *Accepted for publication in MILCOM 2004*, 2004.

[31] Y. Turcotte, O. Tal, S. Knight, and T.R. Dean. Security Vulnerabilities Assessment of the X.509 Protocol by Syntax-Based Testing. *Military Communications Conference 2004*, 2004.

[32] S. Xiao, L. Deng, S. Li, and X. Wang. Integrated TCP/IP Protocol Software Testing for Vulnerability Detection. In *ICCNMC Proceedings of the 2003 International Conference on Computer Networks and Mobile Computing*, page 311, 2003.

[33] S. Xiao, S. Li, X. Wang, and L. Deng. Faultoriented Software Robustness Assessment for Multicast Protocols. *Second IEEE International Symposium on Network Computing and Applications*, page 223, 2003.

[34] S. Zhang, T.R Dean, and S. Knight. Lightweight State Based Mutation Testing for Security. In *Proc TAICPART-MUTATION 2007*, pages 223–232, Windsor, UK, September 2007.