SECURITY VULNERABILITIES ASSESSMENT OF THE X.509 PROTOCOL BY SYNTAX-BASED TESTING

Yves Turcotte
National Defence Headquarters
Ottawa, Canada

and

Oded Tal and Scott Knight
Royal Military College
Kingston, Canada

and

Thomas Dean
Queen's University
Kingston, Canada

ABSTRACT (AbstractID # 904)

This paper describes a methodology for syntax-based vulnerability testing of computer-network protocol implementations, by mutating the protocol data units (PDUs) transmitted to the target implementation.
The implementers of a protocol are under a number of different constraints: time, budget, throughput and memory footprint-size. Adequate attention to secure handling of data structures in a PDU can give way to other pressures. The implementation may be designed to meet conformance-testing cases but can have open vulnerabilities to more obscure cases that might not even be possible during normal operation of the protocol. The vulnerabilities can lead to a compromise of the target's security, e.g. buffer overflow. The vulnerability testing approach described in this paper manipulates the grammar of the targeted network protocol to generate a large number of mutated test-cases that can be used to identify security vulnerabilities.
This work builds on that of Beizer and the PROTOS research group who propose a functional method for assessing protocol implementation security. It adopts a more general approach in its modelling of protocols in order to take advantage of similarities between protocol data structures and to better utilise common abstract syntax constructs (in this case ASN.1), and common transfer syntaxes. It focuses on the mutation of a representation of PDU syntax that is derived from actual protocol PDUs "by example" rather than by specifying and mutating the grammar for the protocol itself. This results in the production of a more universal testing tool applicable to many ASN.1-based protocols with little or no modification. The methodology and tools developed as part of this work were used with success to test a number of network protocols, including a commercial product using ASN.1-specified X.509 public key certificates.

INTRODUCTION

Preface

Conformance test-suites are often developed to ensure that the implementation of a computer-network protocol responds correctly to all known protocol messages and their variations. While protocol specifications and protocol conformance test-suites ensure, to a certain extent, compatibility of interfaces between products, they do not, by themselves, ensure the security of the protocol-implementation.

The implementers of the protocol are under a number of different constraints during the development of their software. The software must be built on time and on budget; it might have to live within a limited memory footprint and be expected to meet demanding throughput performance requirements. Adequate attention to secure handling of data constructions in a Protocol Data Unit (PDU) might give way to other pressures. The implementation may be designed to meet the conformance test-cases but can have open vulnerabilities to more obscure cases, which might not even be possible during normal protocol operation.

Security vulnerabilities can reveal themselves in many forms, such as stack or buffer overflows, which cause the implementation of a theoretically sound protocol to fail. This can enable an attacker to gain privileges or to interfere with the functionality of the system implementing the protocol. The presence of such security vulnerabilities can have serious consequences.

Kaksonen et al. [3] describe a functional method for assessing protocol-implementation security based on protocol syntax, which has been used by the PROTOS research group at Oulu University, Finland. The method extends Beizer's [1] syntax testing principles and produces a large set of carefully crafted protocol data units (PDUs) that are used to test for security vulnerabilities in network protocol-implementations. The security of a protocol-implementation is assessed based on its ability to correctly handle or reject these malformed PDUs and not to reveal security vulnerabilities.

The work described in this paper uses an approach similar to that of PROTOS by testing for security vulnerabilities using malformed PDUs. In this work however, a more general framework and methodology is proposed for the generation of the test PDUs. The new framework can be easily adapted to address a broad range of network protocols.

Objective

The objective of this work is to provide a more general approach to the modelling of protocols, and to syntax-based test-vector generation. The new approach takes advantage of similarities between protocol data structures in order to better utilise common abstract syntax constructs, and common standards for encoding data structures for transmission on a network. The immediate target protocols for the framework are those using ASN.1 [4] to specify their protocol data units. For example, cryptographic keys and user identification information are held in X.509 [6] certificates in a number of PKI (Public Key Infrastructures) communication protocols used to manage security services. The X.509 specification is written using ASN.1. Although the initial focus was on these standards, this work has resulted in the development of a more universal testing-framework, which is applicable without modification to ASN.1-based protocols, and is extensible for encoding rules other than BER or DER [4], which are standard encoding rules specified for ASN.1.

BACKGROUND

Definitions

A **protocol** is an agreed-upon format for transmitting data messages between two devices.

An **abstract syntax** describes the structure of the data, without its semantics, and is not tied to any programming language or implementation platform. **ASN.1** (Abstract Syntax Notation) is an international standard and a formal notation for describing an abstract syntax, which is used in many protocol specifications.

A **concrete syntax** is the representation, in a given programming language, in a given protocol-implementation, of the data structure. That is, it is the way in which the internal data structures of a specific commercial product are organized.

A **transfer syntax** is an agreed-upon bit sequence convention to describe the data structures to be transferred from one protocol-implementation to another. **BER (Basic Encoding Rules)** and **DER (Distinguished Encoding Rules),** which is a subset of BER, are two types of transfer syntaxes associated with ASN.1 for interchanging octets (8-bit bytes) of data. BER and DER follow the format of a Type Length Value triplet (TLV). The Type field is used to encode one of the ASN.1 types (e.g. BOOLEAN, INTEGER, SEQUENCE). The Length field represents the number of octets of the Value field. Finally, the Value field may be a series of octets providing a concrete representation of a value of the basic types, or in the case of constructed types, more TLV triplets.

Syntax testing

Beizer [1] proposes that one formally specify the syntax (grammar) for the protocol language in a convenient notation, such as BNF (Backus-Naur Form). Mutations are then made to the syntactic elements, and the modified grammar is used to produce aberrant test-vectors. Beizer suggests that, because of the large number of possible syntax mutations, automation should play a key role. He also suggests the use of an "anti-parser" in order to compile the BNF grammar to produce "structured garbage." From the protocol syntax, the various fields and field delimiters are identified. Beizer suggests targeting only one field of the input at a time at first, and then to design test-cases with combinations of input.

In the context of syntax testing, two areas can be targeted:

1. Syntax testing of the **concrete syntax** used by the IUT (Implementation Under Test) can be done by modifying the structure and content of the PDU. This can be done even if the specific concrete syntax used is not known. Syntax testing at this level probes the ability of the IUT to handle malformed PDUs that were correctly decoded.
2. Syntax testing of the **transfer syntax** probes the robustness of the IUT encoding/ decoding modules when subjected to incorrectly or unusually encoded PDUs.

The PROTOS approach

The PROTOS work [3] follows the Beizer approach. They have specified the grammar for a number of protocols: WAP, HTTP, LDAP and SNMP. They parsed the grammar specifications to produce a parse-tree representation of the protocol language. They then manipulated the tree and useD the mutated tree to generate thousands of test-cases.

An interesting aspect of the PROTOS approach is that it is directed toward black-box security vulnerability testing and not traditional software quality assurance. The testing technique is not concerned with whether or not the test PDU is processed correctly (as would quality assurance testing). The approach generates thousands of malformed PDUs whose purpose is revealing faults in the handling of input data in the IUT. An IUT is considered to have passed the test if it has rejected the anomalous input without exhibiting such phenomena as crashing, hanging (denial of service), or causing an illegal access to memory. While PROTOS achieved a good level of test execution automation and a certain level of test design automation through scripting, a drawback of this approach is that its

first stage, developing a machine-processable specification of the protocol grammar, is both protocol-specific and labor-intensive.

APPROACH

Overview

The fundamental approach of this work, as depicted in Figure 1, is based on two ideas: 1) General ASN.1 and DER grammars (protocol definitions), which include possible mutations, and 2) Using a mark-up/implement architecture in order to make PDU mutating more efficient.

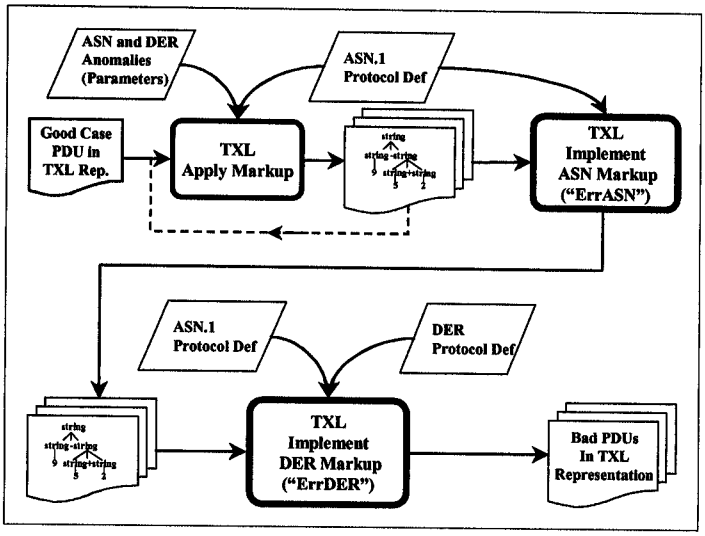

Figure 1. The general approach

Using a valid PDU (collected from legitimate traffic with the IUT) and a general ASN.1 grammar, a parse-tree is obtained and mutated at the abstract syntax level with the help of a transformation tool. Then each mutant PDU can also be mutated when it is encoded at the transfer syntax level based on a general DER grammar. The main tool supporting this approach is TXL, which is described in the next section.

Test-case generation by Protocol-tester is driven by a number of testing strategies specified by PDU mutation rules. The rules specify different kinds of changes to be made to the fields of the PDU based on the type of the field. Mutations can also involve a number of fields (e.g. swapping elements), or structural modifications (e.g. adding or removing sub-components of the PDU). PDU mutation is carried out in two phases:

1. In the first phase an optional error-code is added to each ASN.1 type, in order to specify the nature of the mutation to be performed at the next phase on any given ASN.1 field.
2. In the second phase various mutations are executed on a valid PDU, using the TXL transformation definitions specific to each mutation type.

Libraries of testing strategy rules can be applied to each PDU. Therefore, each legitimate example PDU can be used to generate many test-case PDUs.

TXL

TXL is a functional programming language [2] specifically designed to handle transformation tasks. The basic paradigm of TXL involves transforming input to output using a set of transformation rules that describe by example how different parts of the input are to be changed into output. In this work TXL is used to specify a general ASN.1 grammar used to parse valid PDUs. It is also used in the formulation of transformation rules to describe how to mutate a PDU.

A Simplified Example

Throughout this section the description of the vulnerability-testing framework will be placed in the context of a simple imaginary protocol called the Housing Data Protocol (HDP). HDP is used to communicate information about the residents of the streets in a city, one street at a time. For every street, the required data is {house_number and family_name} of its residents. The ASN.1 specification for an HDP packet might look like this:

```
HDP_Packet ::= SEQUENCE {
        number_of_houses        INTEGER
        houses                  Houses}
Houses ::=      SEQUENCE OF House
House :: =      SEQUENCE {
        house_number            INTEGER
        family_name             VisibleString}
```

Protocol-implementations, which may come from different vendors, might exchange HDP packets that have been encoded using DER as the transfer syntax. For example, let us assume that a specific HDP packet contains the data for only two houses: no. 200- Smith, and no. 300- Stevens. The DER encoding scheme for this packet {2, {{200, Smith} {300, Stevens}}} is represented by nested TLV structures. For example the TLV structures representing the {300, Stevens} portion of the packet would be represented by the following octet encoding:

T1	L1	V1					
		T2	L2	V2	T3	L3	V3
16	13	**2**	2	*300*	**26**	7	*Stevens*

These structures would be embedded as the value field of a more complex structure comprising the whole packet. The **type** fields represent data types: SEQUENCE or SEQUENCE OF (16); INTEGER (2) and STRING (26). The length fields represent the length in bytes of the corresponding TLV triplet data field. The *value* fields represent the data itself. For example, in the last TLV triplet, (T3,L3,V3), the type field is 26 since the family name is a string; the length field is 7, because the name Stevens is a string of 7 characters, and the value field is the name itself, Stevens. Also note, that L2 is 2 because the second house number, 300, is represented by 2 octets.

Pre-processing

The first step in the generation of a set of mutated test PDUs begins with a well-formed legitimate test PDU like the one above. The PDU might be collected from live legitimate traffic from the IUT. The PDU in its binary format and the general ASN.1 grammar are fed into an **ASN.1 Parser**, written in Java. The output file of this program is a TXL-readable presentation of the original PDU. The output of this tool looks like a high-level ASN.1 representation of the packet. Note that the production of this representation does not require a grammar specification for the protocol under test. Next, a TXL program introduces error mark-up tokens into the PDU's structure. Each token is identified by an asterisk (*) followed by a unique field-number. These tokens identify sites were error codes may be introduced in the following steps. For the HDP example above, the output file of ASN.1 Parser complete with the mark-up tokens (in bold) would be as follows:

The marked-up version of the PDU

```
SEQUENCE*1 {
  INT*2 2
  SEQUENCE*3 {
    SEQUENCE*4 {
      INT*5 200
      VisibleString*6 "Smith"}
    SEQUENCE*7 {
      INT*8 300
      VisibleString*9 "Stevens"}
  }
}
```

ASN.1 mark-up and mark-up implementation

Suppose that for a particular test-suite the testing strategy calls for the mutation of the original PDU by removing one of the INT fields at the abstract syntax level. The **TXL Apply Markup** program is used to add this mark-up to the PDU. For example, a markup, **ErrASN remove**, might be applied to field 5 of the HDP packet example.

The **TXL Implement ASN Markup** program will then be used to implement this mutation. In the example, the mutation on field number 5 will produce a mutated PDU, in which the first house number (200) is missing. The marked-up PDU for this mutation command would be:

The PDU with one mutation command

```
SEQUENCE*1 {
  INT*2
  SEQUENCE*3 {
    SEQUENCE*4 {
      INT*5 ErrASN remove 200
      VisibleString*6 "Smith"}
    SEQUENCE*7 {
      INT*8 300
      VisibleString*9 "Stevens"}
  }
}
```

After implementing the command the field is removed in the resulting PDU.

Implementing the mutation command

```
SEQUENCE*1 {
  INT* 2
  SEQUENCE *3 {
    SEQUENCE *4 {
      VisibleString*6 "Smith"}
    SEQUENCE*7 {
      INT*8 300
      VisibleString*9 "Stevens"}
  }
}
```

Other possible mutations at the this level include, among others: changing field order, changing the order of larger composite data structures, including 3 houses in the message without changing the number_of_houses, including illegal characters in the family name, duplicating the same house number and family name, and many combinations of such mutations. All of these mutations are modifications of the PDU at the abstract syntax level.

DER mark-up and mark-up implementation

The framework can also generate mutated test PDUs by making modifications at the transfer syntax level. These mutations effect the encoding of packet fields. Suppose that for a particular test-suite the testing strategy calls for the mutation of the original PDU by changing the value of a TLV length field such that it will not be compatible with the corresponding value field. For the HDP example above consider a mutation of the first length field, L1 (the sequence holding the data {300, stevens}). To do this an additional error mark-up, **ErrDER increase_length**, can be added to field no.7. Now the marked-up PDU (mark-ups for: remove field no. 5, and increase the length of field no.7) would look like this:

The PDU with two mutation commands

```
SEQUENCE*1 {
    INT*2
    SEQUENCE*3 {
        SEQUENCE*4 {
            INT*5  ErrASN remove 200
            VisibleString*6  "Smith"}
        SEQUENCE*7 ErrDER Increase_length {
            INT*8 300
            VisibleString*9 "Stevens"}
    }
}
```

Again, after processing the command field no. 5 is removed in the resulting PDU. Note that the **TXL Implement ASN Markup** tool has performed the high-level, abstract syntax modifications required to perform the removal of field no. 5 as before. Note also that the ErrDER mark-up remains. The encoding level modifications are performed at the next stage of processing.

Implementing the ASN mutation command

```
SEQUENCE*1 {
    INT*2
    SEQUENCE*3 {
        SEQUENCE*4 {
            VisibleString*6  "Smith"}
        SEQUENCE*7 ErrDER Increase_length {
            INT*8 300
            VisibleString*9 "Stevens"}
    }
}
```

Using TXL DER Grammar a tool called the **TXL DER Encoder** is now used to transform the test PDU from ASN.1 format into DER format, including implementing mark-ups. At this stage the program mutates the PDU at the DER level, by increasing the value of the first Length field by one. The output of this program is a mutated PDU in TXL DER format:

T1	L1	V1					
		T2	L2	V2	T3	L3	V3
16	14	**2**	2	*300*	**26**	7	*Stevens*

Note, in the small portion of the PDU that was examined above, the value of byte L1 has been increased in the DER encoding from its correct value, 13, to 14. If the whole PDU were to be examined it could also be seen that the encoding of field no. 5 was missing. Possible mutations at the DER encoding level include, among others: using invalid tags, using negative numbers for the house number and for the numberOfHouses, and many combinations of such mutations. All of these mutations are modifications of the PDU at the transfer syntax level.

Post-processing

Finally, the mutated PDU is transformed by a **Post-Processing Parser**, written in Java, into its binary format, which can now be injected into the implementation of the HDP protocol under test. An example of injection might be to package the binary stream in a network packet and send it to the HDP application for processing. A successful test packet will cause the HDP application to fail.

TEST PROCEDURE

Conducting the syntax testing of a protocol implementation can be broken down into several steps, as illustrated at Figure 2. Note that the bold bordered box labeled Transformation contains the components from figure 1, which have been described in the example above.

Step no. 1: specifying the required mutations

The test engineer creates a script that describes the tests to be performed. These are selected from a library of PDU mutation instructions or created for the specific IUT. **TXL ScriptMaker** interprets the test engineer's mutation instructions and converts them into a set of operations used to generate marked-up PDUs. Typical ScriptMaker commands consist of an ASN.1 target type, an error level (ASN.1 or DER levels), a mutation command and its associated parameter. For example:

1. INT ErrASN {remove} – remove INT fields at the application level (abstract syntax).
2. INT ErrDER {increase_Length} – increase the length field at the DER encoding level (transfer syntax).

For example, the first command when run by ScriptMaker would produce instructions to the **TXL Apply Markup** program to produce a set of new marked-up PDUs such that for each new PDU a different INT field of the original valid PDU was removed. It can be seen that each ScriptMaker command can generate many marked-up PDUs. This is especially true of commands that manipulate multiple fields such as changing field orderings. These commands can have a large number of permutations. In addition to the sets of test-case PDUs that are generated by a single ScriptMaker command, there are also commands that let the test engineer combine two or more commands and generate PDUs with multiple mutations [7]. These combined commands are used with caution to prevent an explosion in the number of test-cases.

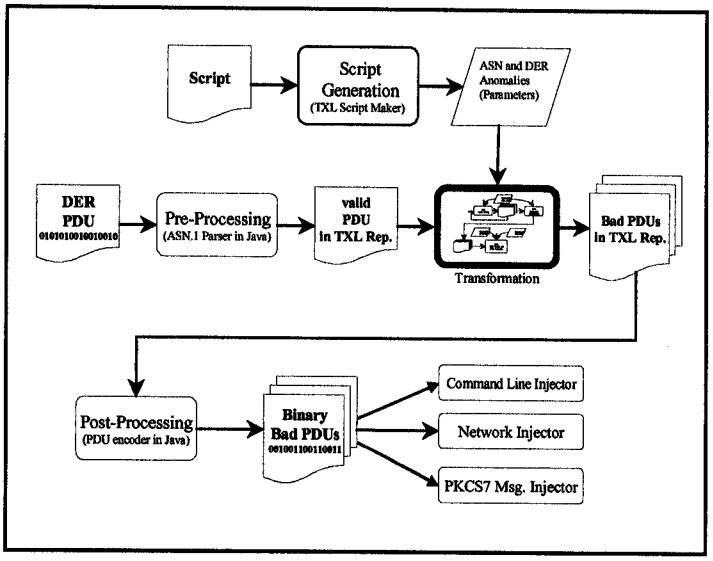

Figure 2. Test procedure

Step no. 2: PDU capture and transformation
A good PDU is captured from legitimate traffic with the IUT. The process described in the simplified example above is used to generate many mutated test-case PDUs. The result is a test-suite of many PDUs in their binary format.

Step no. 3: Injecting and monitoring
The last step test-case injection. Different injectors are used depending on the nature of the IUT. The test injector design can take different forms but always aims at automating the process of sending malformed PDUs to a potentially vulnerable identified interface. For instance, the Network Injector uses the targeted communication protocol to simulate a client-server exchange with a malformed PDU, in the same manner in which a real exchange would take place. Similarly, an injector for some applications is simply a script file containing command lines that execute the IUT application with the various malformed test-case PDUs as parameters.

In association with the injection of each test-case into the IUT there must also be a mechanism for determining if the injection of that test-case caused an error in the IUT. In some cases a diagnostic tool can be used. In other cases some normal, valid activity is requested of the IUT after the injection of each test-case as part of the automated testing process. Logs from the test injector and the monitoring tools are used when as error is detected to identify the PDU that caused the error and the nature of the IUT failure.

MUTATION APPRAOCHES

Brute Force Approach - Application Anomalies
This approach focuses on how the IUT uses the PDU, after it has been DER-decoded, and systematically tests all the ASN.1 types, regardless of how they are actually used for the particular IUT. It relies mainly on the analysis of the definitions of ASN.1 types. A direct benefit from this approach is a wide coverage of all simple and structured fields of the protocol under test while still providing a readily functional test-suite relevant to any ASN.1 derived protocol. Each ASN.1 type was examined and test vectors were designed to address a wide range of syntax errors, e.g. changing the order of its elements, value overflow, etc.

Brute Force Approach - Encoding Anomalies
This approach is also generic in nature (i.e. application-independent) and focuses on anomalies and unusual options of the DER transfer syntax. The resulting test-suite is also applicable to any ASN.1 derived protocol using this transfer syntax. It aims at discovering vulnerabilities with security implications at the DER decoder levels of the IUT. First, examining the TLV structure of the transfer syntax, all type and length fields encoding options of the DER were explored and encoding anomalies were introduced. Then various encoding options of ASN.1 types were explored in order to generate errors in the type, length and value fields, such as a length field, which does not correspond to the length of the value field.

Semantic Augmentation Approach
This approach is based on using some knowledge of the protocol's semantics in order to design test-cases. In general, more test-cases can be defined, depending on the effort placed in the semantic analysis of the protocol and on the purpose and the relationship between various data elements of the PDU. This approach is complementary to

the first two. As tests using this approach are partially semantically driven, and not purely syntax driven, these test strategies must be custom-tailored every time a new protocol is tested for security vulnerabilities. Many field value permutations are covered by the brute force approaches; in this approach field dependencies, protocol options and field particularities are of interest.

TEST RESULTS

Two of the products tested using this approach have been:

1. A certificate manager application for X.509 certificates.
2. Two versions of a commercial PKI product using X.509 certificates (the current release and a previous release).

The test framework did not detect any vulnerabilities in the certificate manager application, but did produce some interesting results for the PKI products.

The framework produced 29,136 test-cases for X.509 Certificates. The test-case injector is a C++ program, based on the PKI's API toolkit, which can be used by an application programmer to communicate with the PKI. Potential vulnerabilities in the earlier version of the product were discovered. The injector caused failures on several test-cases. For both versions of the product test-cases in a third category, although properly handled, took more than 20 minutes to process.

Establishing the severity of the potential failures was not part of this work. However, even without further investigation these failures present denial of service attack vulnerabilities. They may however be indications of more serious security vulnerabilities. These results demonstrate the value of the framework and the testing approach in uncovering security vulnerabilities in protocols defined using ASN.1 and using DER as the transfer syntax.

CONCLUSION

The efficiency of protocol testing for vulnerabilities through syntax testing has clearly demonstrated its strength in the past through the work of PROTOS. This work goes further by introducing a complementary approach to the problem of protocol testing such that the following advantages are achieved:

1. It adopts a more general approach in its modelling of protocols in order to take advantages of similarities between protocol data structures and to better utilise common abstract syntax constructs, and common transfer syntaxes. This resulted in the production of a more universal testing tool applicable to many ASN.1-based protocols with little or no modification and extensible for other encoding rules than DER.
2. It focuses on the mutation of a representation of PDU syntax, derived from actual PDUs "by example" rather than by specifying and mutating the protocol's grammar. By careful choice of example PDUs, to cover various PDU options, the thoroughness of syntax mutation achieved is believed to be as good as what could be obtained through straight mutation of the grammar.
3. The approach does not necessitate the significant conversion process of an ASN.1 specification into a machine-readable grammar each time a new protocol is tested. Instead, the framework translates the initial example PDU into a concrete syntax in TXL, which is very similar to the original representation in ASN.1.
4. The two-phase approach to the error transformation process makes it easy to support more sophisticated testing strategies. The two-phases: applications of error mark-ups, and the implementation of error mark-ups, are cleanly separated. The implementations of the mark-ups tend to be more primitive. A broad range of more sophisticated testing strategies can be implemented by the specification of a group of more primitive mark-ups.

REFERENCES

1. B. Beizer, Software Testing Techniques, 2nd Edition, Van Nostrand Reinhold, New York, 1990.
2. J. Cordy, The TXL Programming Language ver. 10, http://www.txl.ca/nabouttxl.html, 2000.
3. Kaksonen, R., Laasko, M. and Takanen, A., Vulnerability Analysis of Software through Syntax Testing, 2000. http://www.ee.oulu.fi/research/ouspg/protos/analysis/WP2000-robustness/index.html, 2001.
4. Dubuisson, O., "ASN.1 Communication between Heterogeneous Systems", Academic Press, San Diego, 2001.
5. PKCS#7 - Cryptographic Message Syntax Standard. http://www.rsasecurity.com/rsalabs/pkcs/pkcs-7/, RSA Data Security, Inc, 2004.
6. Public-Key Infrastructure (X.509) (pkix). http://www.ietf.org/html.charters/pkix-charter.html, 2004.
7. Turcotte, Y., Tal, O., Knight, S. and Dean, T. "Universal methodology and tools for syntax-based vulnerability testing of protocol implementations". Submitted to the Journal of Computer Security, February 2004.