A Formal Approach to Robustness Testing of Network Protocol

Chuanming Jing[1,3], Zhiliang Wang[2,3], Xia Yin[1,3], and Jianping Wu[1,2,3]

[1] Department of Computer Science & Technology, Tsinghua University
[2] Network Research Center of Tsinghua University
[3] Tsinghua National Laboratory for Information Science and Technology, Beijing, P.R. China, 100084
{jcm,wzl,yxia}@csnet1.cs.tsinghua.edu.cn, jianping@cernet.edu.cn

Abstract. Robustness testing of network protocol aims to detect vulnerabilities of protocol specifications and implementations under critical conditions. However, related theory is not well developed and prevalent test practices have deficiencies. This paper builds a novel NPEFSM model containing sufficient inputs and their processing rules to formalize complex protocol. Based on this model, Normal-Verification Sequence is proposed to enhance verdict mechanism. We adopt various strategies to generate anomalous values for some fields of messages and further apply pairwise combination to systematically mutate messages. We propose compound anomalous test case to simplify test sequences and give its generation algorithm. Standard test specification language TTCN-3 is extended to describe compound anomalous test cases. As a case study, we test OSPFv2 sufficiently with a test system based on extended TTCN-3. Our method and test system can effectively discover vulnerabilities of protocol implementations as well as their inconsistencies with specifications.

1 Introduction

Network protocols are often partially specified. There are numerous invalid inputs for protocol implementations and how to handle these inputs is usually unspecified or specified ambiguously in protocol. Also protocol specifications contain optional requirements specified by "MAY" statements. These two cases provide certain flexibility to protocol implementations. As conformance testing only verifies whether an implementation conforms to its specification or not, the capability of error-detection is limited. National Vulnerability Database [1] reports that about 60% of the software vulnerabilities detected in 2007 were caused by input validation, format string vulnerability and buffer errors. Protocol robustness testing is the test to verify whether IUT (Implementation under Test) can function correctly in the presence of invalid inputs or stressful environmental conditions [2]. Robustness testing aims to detect vulnerabilities of protocol specifications and implementations, including [3]: vulnerabilities of malformed message parsing; vulnerabilities of state transitions; hole of buffer overflow etc. There have been large previous works about robustness testing [4-14]. Although their test practices have found vulnerabilities of protocol implementations, these approaches have certain limitations: 1) test cases generation lacks guidance of

J. Cao et al. (Eds.): NPC 2008, LNCS 5245, pp. 24–37, 2008.

theory; 2) verdict mechanism needs improvement; 3) structure of test case is not optimal, resulting in large test costs; 4) test system is not generic to other protocols; 5) most use programming languages (e.g. C) to build test suite, so the readability, extensibility and maintainability of test suite are not good.

To cope with these deficiencies, we build a novel Nondeterministic Parameterized Extended Finite State Machine model. Our model has distinct benefits: 1) it contains sufficient inputs and their processing rules, thus it can guide robustness testing; 2) it supports variables, parameters of inputs/outputs and operations based on these values, thus it can model complex protocols. Based on this model, Normal-Verification Sequence is proposed to enhance verdict mechanism. Data-driven robustness testing focuses on inputting various invalid messages. We adopt various mutation strategies to generate anomalies for some fields of messages and further apply pairwise strategy [15,16] to systematically import anomalies to mutate multiple fields for each message. To inject test data efficiently and effectively, compound anomalous test case is proposed to simplify test sequences. The algorithm of test case generation is given. TTCN-3 [20] is extended to describe compound anomalous test cases. Implementing the function of generating test data automatically, the extended description is very simple and convenient to use. As a case study, we test OSPFv2 [21] sufficiently with a test system based on extended TTCN-3.

The rest of this paper is organized as follows. Related works are introduced in section 2. Section 3 proposes NPEFSM model. In section 4, pairwise strategy is used to combine anomalies and test case generation is discussed. We apply TTCN-3 to robustness testing and extend it in section 5. In section 6, we test OSPFv2 using our method. Conclusions and future work are given in section 7.

2 Related Works

Related works can be classified into research on model-based robustness testing [4,5] and test practices (often called Fuzz testing [6-14]). The model and framework for robustness testing are not well developed. [4,5] propose a formal framework, but mutation operations and fault injections are not done automatically. Hence it is difficult to generate large number of test cases.

Fuzz testing is a black-box testing method by injecting faults. The procedure is to generate test data, inject test data to IUT and make verdict. Currently, there are two methods to obtain numerous invalid messages: designing manually using script language [6-12]; generating semi-randomly data (e.g. most of tools listed in [13]). Various script languages are used to describe invalid messages such as BNF (Backus-Naur Form) [6], SBNF (Strengthened BNF)[7,8], SCL (Semantic Constraint Language) [9,10] and XML [11,12]. The production and injection of invalid PDUs are all done by tools implemented by C or Java. Also, other Fuzzing tools [13] can produce semi-random messages which are often blind to testing and each tool can only test a certain protocol due to weak extensibility. Intelligent Fuzzing requires injecting invalid inputs on the corresponding state. [7,8,11] all propose state identification by inferring from I/O sequences logged, but it is not practical for complex protocols. Related works about verdict mechanism are also not well developed. In test practice, they observe whether IUT is crashed or monitor the performance (e.g. CPU usage) of IUT

under invalid injections. In [10,11], a simple sequence consisting of a valid request and corresponding reply is used to make verdict after fault injection.

So, it is highly desirable to have a formal approach to robustness testing. In our previous work [17], single-field mutation testing is studied. Based on this work, we give our full solutions for protocol robustness testing in this paper.

3 Formal Model and Testing Framework

3.1 Model Definition

Usual protocol specifications include variables and operations based on variable values. Extended Finite State Machine (EFSM) can be used in this situation. However, it is still not powerful enough to model some protocol systems where there are parameters associated with inputs and have effects on the predicates and actions of transitions [18]. Hence Parameterized Extended Finite State Machine (PEFSM) [18] is used to model protocol specifications. Robustness testing requires injecting many invalid messages. As most invalid messages and their processing rules are not well prescribed, state transitions after these invalid injections are often nondeterministic. So, we build a model for protocol robustness testing using NPEFSM (Nondeterministic Parameterized Extended Finite State Machine). Our model covers more detailed and precise nondeterministic features than traditional nondeterministic FSM and EFSM model.

Definition 1: NPEFSM for Protocol Robustness Testing
A Nondeterministic Parameterized Extended Finite State Machine (NPEFSM) is a 6-tuple $M=<I, O, \vec{X}, S, s_0, T>$, where:

1. $I=\{ i_1(\vec{v_1}), i_2(\vec{v_2}),...,i_p(\vec{v_p}) \}$ is the input alphabet with parameters $\vec{v}$; each input symbol $i_k(\vec{v_k})$ $(1\le k\le p)$ carries a vector of parameter values $\vec{v_k}$;

   Also, we define $I=I_{spec}\cup I_{unspec}$. I_{spec} includes inputs that are prescribed in protocol specification and composed of valid PDUs as well as some invalid PDUs. I_{unspec} includes numerous inputs that are not prescribed definitely in protocol specification and composed of various invalid PDUs.
2. $O=\{o_1(\vec{w_1}), o_2(\vec{w_2}),...,o_q(\vec{w_q})\}$ is the output alphabet with parameters $\vec{w}$; each output symbol $o_k(\vec{w_k})$ $(1\le k\le q)$ carries a vector of parameter values $\vec{w_k}$.
3. $\vec{X}$ is a vector denoting a finite set of variables with default initial values.
4. S is a finite set of states, $S=S_{spec}$, S_{spec} includes states prescribed in protocol specification. We introduce and define $S_?=\{s_{?i} \mid i=1,2,\ldots\}$, $s_{?1}$, $s_{?2}$…are all nondeterministic states after nondeterministic or undefined transitions but within a range of states according to corresponding ambiguous protocol specification, i.e. for $i=1, 2,\ldots$, $s_{?i} \in S_i \subseteq S_{spec}$. So, $S = S_{spec} = S_{spec} \cup S_?$.
5. s_0: initial state.
6. T: a set of transitions. For $t\in T$, $t = s \xrightarrow{i(\vec{v})/o(\vec{w})/P(\vec{X},i(\vec{v}))/A(\vec{X},i(\vec{v}),o(\vec{w}))} s^*$ $(s\in S, s^*\in S)$ is a transition where s and s^* are the starting and ending state of this transition respectively; $i(\vec{v})/o(\vec{w})$ is the input/output with parameters; $P(\vec{X},i(\vec{v}))$ is a predicate of the variables and input parameters; the action $A(\vec{X},i(\vec{v}),o(\vec{w}))$ is an operation on variables

as well as output parameters and this operation is based on current variable values and input parameter values.

$T=T_{deter}\cup T_{nondeter}=T_{deter}\cup(T_{nondeter\text{-}spec}\cup T_{nondeter\text{-}unspec})$. Each transition of T_{deter} is uniquely deterministic in protocol. $T_{deter}=\cup t$: $s_j \xrightarrow{i_j(\vec{v})/o_j(\vec{w})/P(\vec{X},i_j(\vec{v}))/A(\vec{X},i_j(\vec{v}),o_j(\vec{w}))} s_k$, where $s_j\in S_{spec}$, $s_k\in S_{spec}$; $i_j(\vec{v})\in I_{spec}$; $o_j(\vec{w})\in O$ *or* $o_j(\vec{w})=Null$. Each of $T_{nondeter\text{-}spec}$ is nondeterministic but specified clearly in protocol specification. $T_{nondeter\text{-}spec}=\cup t$:$s_j \xrightarrow{i_j(\vec{v})/o_j(\vec{w})/P(\vec{X},i_j(\vec{v}))/A(\vec{X},i_j(\vec{v}),o_j(\vec{w}))} s_{?k}$, where $s_j\in S_{spec}$, $s_{?k}\in S_?$; $i_j(\vec{v})\in I_{spec}$; $o_j(\vec{w})\in O$ *or* $o_j(\vec{w})=Null$. Each of $T_{nondeter\text{-}unspec}$ is nondeterministic and unspecified or specified ambiguously in specification. $T_{nondeter\text{-}unspec}=\cup t$: $s_j \xrightarrow{i_j(\vec{v})/-/true/-} s_{?k}$, where $s_j\in S_{spec}$, $s_{?k}\in S_?$; $i_j(\vec{v})\in I$, output and action are unspecified or specified ambiguously. □

Figure 1 shows a part of NPEFSM for OSPFv2 [21] Neighbor State Machine, predicates and actions are omitted. An example of transition is given in Table 1. The inputs and outputs are parameterized Database Description Packets (DDP). The parameters of DDP are DD sequence number (denoted as *Seq*) and I/M/MS (denoted as *Ims*). Predicate includes sequence number checking, I/M/MS checking and other validations. *y* is a variable used to check sequence number.

Figure 2 shows two kinds of state transitions under invalid injections. Figure 2(a) shows that after receiving an invalid input i_k ($i_k\in I_{spec}$), s_i transits to s_j according to related description in protocol specification. For example, OSPFv2 specification prescribes that receiving duplicate DDP will trigger "SeqNumberMismatch" and transit the state from "Exchange" or higher to "Exstart". Figure 2(b) shows that s_i transits to state $s_{?k}$ ($s_{?k}\in S_k=\{s_{ki}\mid i=1,2,\ldots\}\subseteq S_{spec}$) because the transition after receiving i_j ($i_j\in I$) is prescribed indeterminately, ambiguously or even not prescribed in protocol specification. For example, during Database Exchanging, after receiving DDP, whether to check the syntax of each field of DDP.LSAHeader is not specified definitely. Suppose this syntax error: LSAHeader.LinkStateID="FFFFFFFF", if IUT checks this field, event "SeqNumberMismatch" will be triggered to transit the state to "Exstart". Otherwise, exchanging will go on until this fault can be found (maybe in LSA requesting process). So, after receiving i_j(DDP), $s_{?k}\in S_k=\{s_{k1}$="Exstart", s_{k2}="Exchange"$\}$ and $t_j\in T_{nondeter\text{-}unspec}$.

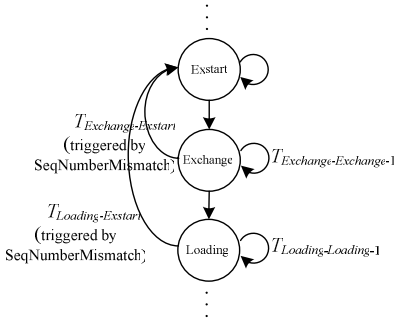


Fig. 1. A Part of NPEFSM for OSPFv2 Neighbor State Machine: Database Exchange

Table 1. An Example of Transition

Name	$T_{Exchange\text{-}Exchange\text{-}1}$
Start State	Exchange
End State	Exchange
Input	DDP(*Seq1, Ims1*)
Output	DDP(*Seq2, Ims2*)
Variables	*y*,…
Predicate	(*Seq1*==*y*)&& (*Ims1*==011)&&…
Action	*Seq2*=*y*; *y*=*Seq1*+1;...

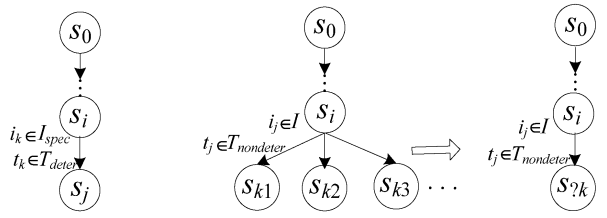


Fig. 2. (a) State transition prescribed definitely in protocol specification after invalid input **(b)** State transition prescribed ambiguously, indeterminately, or even not prescribed in protocol specification after invalid input

Some "MAY" statements in protocol specification also specify transitions belonging to $T_{nondeter\text{-}spec}$. We omit further exemplification.

Some special inputs can transit each of several states (denoted as S') to the same state, we propose Forced Transition as follows:

Definition 2: Forced Transition

Let $S' \subseteq S_{spec}$ *and* $s_j \in S_{spec}$.

$(\forall s \in S') \rightarrow s_j$ *is a Forced Transition, iff,* $\exists\ i_j(\vec{v}) \in I_{spec}$, *such that* $\forall s \in S'$, $s \xrightarrow{i_j(\vec{v})/-/true/-} s_j$.

Especially, if $s_j = s_0$, such a forced transition is also called **Reset Transition**.

In Figure 3, if Forced Transition exists, $s_{?k}$ can receive certain input and transit to a deterministic state s_j according to Definition 2. In the former example about OSPFv2 Data Exchanging, event "SeqNumberMismatch" can force the machine to transit from $s_{?k}$ to s_j (s_j ="Exstart"), whether $s_{?k}$ ="Exstart" or "Exchange".

3.2 Robustness Requirement and Normal-Verification Sequence

The structure of a test case in conformance testing can be described as follows: **Test Case**=<State Leading Sequence, Executing Sequence, State Verification Sequence> [17, 18]. In robustness testing, we introduce a term called "anomalous test case" which can inject invalid data on corresponding state and make verdict. Instead of State Verification Sequence of conformance testing, Normal-Verification Sequence of robustness testing is executed to verify whether the state machine works properly. If it returns "Fail", we conclude that IUT behaves abnormally and has poor robustness. The structure of an anomalous test case can be described as follows: **Anomalous Test Case**=<State Leading Sequence, Invalid PDU Inputting, Normal-Verification Sequence>. The first step of robustness testing is to construct robustness requirement. In this paper, we use an intuitive and practical robustness requirement that IUT must keep normal state and continue normal operations conforming to protocol specification under invalid injections. According to the state transitions shown in Figure 2, we propose two types of Normal-Verification Sequences as follows:

Normal-Verification Sequence_1

Suppose at state s_i, an invalid PDU $i_j(\vec{v})$ is received and $i_j(\vec{v}) \in I_{spec}$.

If t: $s_i \xrightarrow{i_j(\vec{v})/o_j(\vec{w})/P(\vec{X},i_j(\vec{v}))/A(\vec{X},i_j(\vec{v}),o_j(\vec{w}))} s_j$, i.e. $t \in T_{deter}$. *Normal-Verification Sequence= state verification sequence of s_j.*

Normal-Verification Sequence_2

Suppose at state s_i, an invalid PDU $i_j(\vec{v})$ is received and $i_j(\vec{v}) \in I$.

If t: $s_i \xrightarrow{i_j(\vec{v})/o_j(\vec{w})/P(\vec{X},i_j(\vec{v}))/A(\vec{X},i_j(\vec{v}),o_j(\vec{w}))} s_{?k}$ *or t:* $s_i \xrightarrow{i_j(\vec{v})/-/true/-} s_{?k}$, i.e. $t \in (T_{nondeter\text{-}spec} \cup T_{nondeter\text{-}unspec}) = T_{nondeter}$. *Normal-Verification Sequence = state identification sequence of $s_{?k}$.*

State identification [18][19] will cause the robustness test sequences to be very complex, so we define another Normal-Verification Sequence to approximately replace Normal-Verification Sequence_2:

Normal-Verification Sequence_2-1

After receiving invalid PDU, the ending state transited to is $s_{?k} \in S_?$ and $s_{?k} \in S_k = \{s_{ki} \mid i=1,2,\ldots\} \subseteq S_{spec}$, If there exists Forced Transition: $(s_{?k} \in S_k) \rightarrow s_j$, *Normal-Verification Sequence* $\approx ((s_{?k} \in S_k) \rightarrow s_j)$ + (*state verification sequence of s_j*).

Forced Transition and Reset Transition are very common in network protocols. For OSPFv2, event "1-Way", "KillNbr", "SeqNumberMismatch" and "BadLSReq" can trigger Forced transition. So Normal-Verification Sequence_2-1 can be widely used.

A PEFSM can be unfolded into a FSM. In this paper, we do not discuss the construction of state verification sequence for FSM or PEFSM model which is a classical problem in conformance testing [18]. We further illustrate our robustness requirement. For anomalous test cases using Normal-Verification Sequence_1, robustness requirement is the same with conformance testing. For test cases using Normal-Verification Sequence_2-1, robustness requirement can be illustrated using a Probabilistic Finite State Machine (PFSM) shown in Figure 4. s^* is a trap state which is not an element of S_{spec}. P_1 is the probability of state transition from s_i to s^* after receiving an invalid input. P_2 is the probability of state transition from s^* to s^* after receiving an input which is used to trigger Forced Transition. Strict robustness requirement discussed before is P_1=0. In order to ease test practices, we adopt $P_1 \cdot P_2$=0 as our robustness requirement.

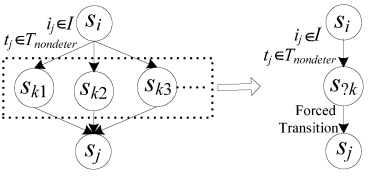


Fig. 3. Forced Transition of $s_{?k}$

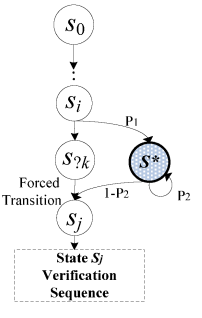


Fig. 4. Robustness requirement analysis

4 Test Case Generation

4.1 Invalid Inputs Generation

A PDU is composed of several fields denoted as f_1, f_2…Invalid PDUs can be syntactically or semantically invalid. The former disobey protocol specification data formats. The latter have valid syntax but conflict with protocol state, configuration and policies. We mutate several valid inputs to generate many messages either syntactically invalid or semantically invalid.

Single-field mutation rules: $Fun_{FieldMutation}()$

We define some typical invalid field values which attackers tend to exploit:

1) Field value mutation rules

In [17], we define Boundary value; Field values mismatch; Format error; Length, Checksum and Encapsulation error. We also define input partition values: {Min, Min+(Max-Min)/n, Min+2*(Max-Min)/n,…,Max}(Suppose Field $f_i \in$ (Min,Max)). n is a parameter set by tester. All these values can be used to replace a valid field value.

2) Field mutation rules:

- Removal and Addition: a field of PDU is removed or added;
- Overflow: one field is replaced with another field with bigger bytes;
- Permutation: sequence of fields in PDU changes.

Multi-field mutation rules using pairwise algorithm

Some fields of one message may have consistency with each other. Value changing of one field may influence values of other fields. Also, protocol implementations may not parse fields of receiving message in the sequence of one by one. So it is necessary to inject messages whose multiple fields are invalid. Suppose 8 fields of one PDU will be mutated and each field has 5 invalid values, total of mutated PDUs will be 5^8=390625. So exhaustive testing is impractical. In this paper we introduce pairwise combination which can guarantee that each pair of faults between any two fields is covered by at least one combination. Pairwise combination is a good trade-off between test effort and test coverage. Algorithm 1 adopts a heuristic pairwise algorithm called In-Parameter-Order [15,16] to generate test data.

Algorithm 1. pairwise (F, Q)

Input: $F=\{f_1, f_2,\ldots, f_n\}$; $Q=\{ q_{f_1}, q_{f_2},\ldots, q_{f_n} \}$; /*each q_{f_i} is a set of values for field f_i. */

Output: $T=\{T_1,T_2,\ldots T_m\}$; /*each $T_i=\{v_{f1}, v_{f2},\ldots,v_{fn}\}$, where $v_{f_j} \in q_{f_j}$ ($1 \le j \le n$), i.e. T_i is a n-dimension vector containing values for field $f_1, f_2,\ldots, f_n$, respectively.*/

```
1  T={(v1, v2) | v1and v2 are values of f1and f2 ,respectively};
2  if (n ==2) return T;
3  for each field fi, i=3,4,…,n do
4     T= In-Parameter-Order (T, Q, fi);  /*see references [15,16] */
5  return T;
```

Then we give an example. Suppose $F=\{f_1, f_2, f_3, f_4\}$, each has four unsigned boundary values and one 2-partition values. $q_{f_1} = q_{f_4} = \{00,01,7F,FE,FF\}$; $q_{f_2} = \{0000,0001,7FFF,FFFE,FFFF\}$; $q_{f_3} = \{000000, 000001, 7FFFFF, FFFFFE, FFFFFF\}$. Applying Algorithm 1, we can get test data shown in Figure 5:

$$T_{pairwise}(F(f_1,f_2,f_3,f_4))=\begin{pmatrix} T_1 \\ T_2 \\ T_3 \\ T_4 \\ \cdots \\ T_{32} \end{pmatrix} = \begin{vmatrix} f_1 & f_2 & f_3 & f_4 \\ 00 & 0000 & 000000 & 00 \\ 00 & 0001 & 000001 & 01 \\ 00 & 7FFF & 7FFFFF & 7F \\ 00 & FFFE & FFFFFE & FE \\ 00 & FFFF & FFFFFF & FF \\ 01 & 0000 & 000001 & 7F \\ 01 & 0001 & 000000 & FE \\ \cdots & \cdots & \cdots & \cdots \\ 00 & FFFF & 000000 & 7F \end{vmatrix}$$

Fig. 5. Test generation using pairwise combination

In above example, compared to the exhaustive combination (test data set total: 5^4=625), Algorithm 1 only generates 32 test data and each pair of faults between any two fields is covered by at least one PDU.

4.2 Robustness Test Case Generation

According to the definition of robustness testing (see section 1), the quantity and variety of invalid messages are important criterions for testing. There are numerous invalid PDUs and how to inject needs to be well studied. So we propose compound anomalous test case which can also simplify test sequences: 1) One compound anomalous test case focuses on several fields of one PDU (the combination of these fields can be denoted as $F_l \subseteq PDU$). If the verdict is "Fail", it means IUT cannot parse F_l with robustness; 2) Two or more anomalous messages with different invalid values of F_l should be injected in one compound anomalous test case. Values of other fields of these messages cannot be mutated and keep valid.

If $|F_l|=1$, it means only one field is mutated and the corresponding test case belongs to single-field robustness testing. If $|F_l|\geq 2$ (means multiple fields are mutated), this test case belongs to multi-field robustness testing. For test case returning "Fail" in multi-field testing, we should decompose it into several separated test cases executed further to analyze why it fails. As robustness testing is often done after conformance testing, the pass rate is often high, thus test execution of separated test cases will not consume much work. For test cases using Normal-Verification Sequence_1, the formats of Compound Anomalous Test Cases are constructed as follows:

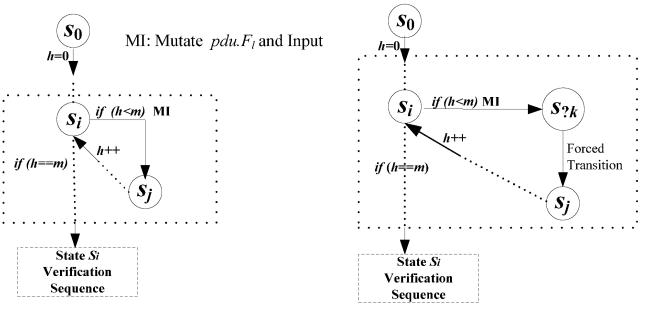
Compound Anomalous Test Case-1(m, $pdu.F_l$) = < State (s_0 to s_i) Leading Sequence, {Invalid *pdu* Inputting, State (s_j to s_i) Leading Sequence}**m*, State Verification Sequence of s_i >.

Where, "{}**m*" means the sequences contained in "{}" are executed *m* times. Also, the fields under test must be mutated before each executing loop. *m* means the number of invalid messages generated by mutation rules for $pdu.F_l$.

For test cases using Normal-Verification Sequence_2-1, the formats of Compound Anomalous Test Cases are constructed as follows:

Compound Anomalous Test Case-2 (m, $pdu.F_l$) = <State (s_0 to s_i) Leading Sequence, {Invalid *pdu* Inputting, State($s_{?k}$ to s_j)Leading Sequence (suppose Forced Transition ($s_{?k} \in S_k$)$\rightarrow s_j$ exists), State(s_j to s_i) Leading Sequence }*m, State Verification Sequence of s_i >.

Figure 6 shows structures of compound anomalous test cases. Algorithm 2 is defined for Compound Anomalous Test Case-1 generation. We omit the algorithm of Compound Anomalous Test Case-2 generation due to space limitation.



(a) Compound Anomalous Test Case-1 (b) Compound Anomalous Test Case-2

Fig. 6. Two Types of Compound Anomalous Test Cases ($pdu.F_l$ is under test)

Algorithm 2: Compound Anomalous Test Case-1 generation

Input: PDU $pdu=\{f_1, f_2, \ldots, f_n\}$; /* *pdu: a valid PDU* */
$F_l \subseteq pdu$; /* F_l *:a fields set under test* */
s_i, s_j;/**inject anomalies on state* s_i *and the state transits to* s_j *in protocol specification**/
Output: $TestCase_{F_l}$; /* *test case for* $pdu.F_l$ */

Initial Value: $TestCase_{F_l}$ =**Null**;

1 $Q=\{ q_{f_1}, q_{f_2}, ..., q_{f_n} \}$; $q_{f_1}, q_{f_2}, ..., q_{f_n}$ =**Null** ; /* q_{f_i} *is a set of values for field* f_i */

2 $T_{F_l} = \{T_1, T_2, \ldots T_m\}$; $T_1, T_{2,\ldots}$ T_m=**Null**; /* *see Algorithm* 1 *in section* 4.1*/

3 **For each** $f_k \in F_l$ **do**

4 $q_{f_k} = Fun_{FieldMutation}(f_k)$; /**generate anomalous values, see section* 4.1*/

5 **If** ($|F_l|$=1) /**single-field robustness testing**/

6 $T_{F_l} = Q$;

7 **If** ($|F_l| \geq 2$) /**multi-field robustness testing**/

8 T_{F_l} = pairwise (F_l, Q); /* *see Algorithm* 1 *in section* 4.1*/

9 $TestCase_{F_l}$ **.add(**State (s_0 to s_i)Leading Sequence);

10 **For each** $T_h \in T_{F_l}$ **do**

11 **replace** each field of $pdu.F_l$ **with** values in T_h, respectively;

12 $TestCase_{F_l}$ **.add (**Invalid *pdu* Inputting**,** State (s_j to s_i) Leading Sequence**)**;

13 $TestCase_{F_l}$ **.add(**State Verification Sequence of s_i**)**;

14 **Return** $TestCase_{F_l}$;

Compound Anomalous Test Case-1 is similar to test case of conformance testing and we do not discuss its property. For Compound Anomalous Test Case-2, different invalid PDU is injected in each loop. We have deduced that the "Fail" probability for each loop is $P_1 \cdot P_2$ (see section 3.2 and Figure 4).Then the "Fail"probability of Compound Anomalous Test Case-2 can be deduced:

$P_{fail}(m) = 1 - \prod_{i=1}^{m} (1 - P_{i\_1} \cdot P_{i\_2})$ (where $P_{i\_1} \cdot P_{i\_2}$ is the fail rate for *ith* invalid injection)

From this equation, we can deduce that the fail rate increases as *m* increases. An intuitive reason is that the more invalid data are injected, the more holes may be discovered.

5 TTCN-3 Extensions and Test System

TTCN-3 [20] is developed by ETSI (European Telecommunications Standards Institute) and standardized by the ITU-T. Test suite described by TTCN-3 has good readability, extensibility and maintainability. But TTCN-3 is not flexible: 1) TTCN-3 cannot support mutation operations well, so we should define thousands of invalid test data using TTCN-3 [17]; 2) Using TTCN-3, it is difficult or even impossible to give the description of compound anomalous test cases.

We extend TTCN-3 in syntax for robustness testing as follows: 1) Add a new keyword "pairwise" which represents the complex algorithm of compound anomalous test case generation; 2) Add a new keyword "count", the value behind this keyword means the number of mutations for one field; 3) In the description statement using "pairwise", each field of the message in format of "global template" can be modified to be invalid so test suite need define only a little data.

If the parameters of "pairwise" statement are only for one field of one message, the test case belongs to single-field testing. Otherwise, it belongs to multi-field testing and pairwise combination will be used. Similar to extension using "pairwise", we can make extensions to accomplish other mutations rules (e.g. Removal, Overflow) defined in section 4.1. We omit these due to space limitation. "pairwise" statement supports describing two types of Compound Anomalous Test Cases. We give a description of test case belonging to Compound Anomalous Test Case-2:

```
testcase Onebyone_HL1_Opt( )  runs on MyTestComponentAsync
  system SystemComponent   {
    map(mtc:MyPortAsync, system:SystemPort1);
    P1( );
    pairwise HL1( Mask octetstring 4 count 5 record_value;
                  Hint octetstring 2 count 6;
                   Opt octetstring 1 count 8) {
       Onebyone_HL1();
    }
    Normal_Verification( );
   }
```

Most anomalies are generated automatically according to $Fun_{FieldMutation}()$ (see section 4.1). "**octetstring** n **count** m" means generating invalid data set which consists of

4 boundary unsigned values and m-4 partition values (see section 4.1), each is in the format of n octetstrings. Some anomalies can be defined manually in "Const Record" (e.g. "record_value" in above statements). When above statements (other keywords in bold are introduced in [20]) are compiled, invalid data for field HL1.Mask, HL1.Hint and HL1.Opt are generated (e.g. $q_{f_{Mask}}$ ={4 boundary values, (5-4) partition values and values in "record_value"}), further pairwise algorithm $T_{pairwise}(F(f_{Mask}, f_{Hint}, f_{Opt}))$ is executed to generate invalid messages. "Onebyone_HL1" is a function using Forced Transition. During test case executing, "Onebyone_HL1" will be executed continually for many loops until all the invalid test data are injected to IUT. Finally, Normal_Verification() is executed to make verdict.

Based on TTCN-3 and its extension, we have developed a test system called PITSv3 which is introduced sufficiently in [23]. We omit this due to space limitation.

6 Case Study: OSPFv2

OSPFv2 [21] defines five kinds of messages including: Hello, Database Description (DDP), Link State Request (LSR), Link State Update (LSU) and Link State Acknowledgment (Ack). It also defines five kinds of Link State Advertisements (LSAs). We design test suites for single-field and multi-field robustness testing respectively. We use Forced Transitions (e.g. "1-Way", "SeqNumberMismatch" and "BadLSReq" [21]) to construct Normal-Verification Sequence.

Table 2 lists test suite. Invalid messages are injected on corresponding state. In test practice, we use PITSv3 to connect with IUT through a link. We choose Zebra-0.94 [22] installed in Linux as IUT.

Table 2. Test suite of compound anomalous test cases for OSPFv2

Test Group	Test Content (State / Invalid PDU received)	Number	
		Single-field	Multi-field
OSPF Head	Init / Hello	14	8
Hello	2-way / Hello	16	6
DDP0	Exstart / DDP0(without LSA Header)	8	4
DDP1	Exchange / DDP1(include one LSA Header)	22	12
LSR	Exchange / LSR	6	2
Ack	Exchange / Ack	16	6
LSA_HEAD	Exchange / LSU(include Router_LSA)	14	10
LSU_RLSA	Exchange / LSU(include Router_LSA)	16	10
LSU_NLSA	Full / LSU (include Network _LSA)	4	5
LSU_S3LSA	Full / LSU (include Type3 Summary _LSA)	4	1
LSU_S4LSA	Full / LSU (include Type4 Summary _LSA)	4	1
LSU_AsLSA	Full / LSU (include As External_LSA)	6	2
	Total	130	67

For single-field testing, we set *m* with different values using "count" keyword in TTCN-3 (see section 4.2 and 5). During test, we choose *m*=5~30. Thus, about 650~3900 invalid messages are injected in total 130 test cases. Test results are shown in Figure 7 (Fail Rate I). Test verdicts of Fail Rate I base on robustness requirement (see section 3.2). Fail rate is 5.38% when *m*=5~20 and it increases to 6.92% when

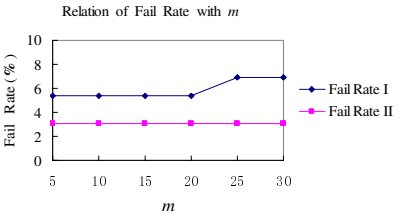


Fig. 7. Test results for single-field robustness testing of Zebra-0.94

m=25, 30. These results indicate that the more invalid data are injected, the more holes may be discovered (see section 4.2). During test, we also sum up test cases (Fail Rate II) returning "fail" due to critical vulnerabilities which cause IUT to crash or be quiescent (IUT may not crash, but can not reply to inputs). Fail Rate II shows that there are always 4 test cases (fail rate: 3.08%) returning "Fail" due to critical vulnerabilities as *m* increases from 5 to 30.

In multi-field testing, we assign *m*=5 (each field has 5 invalid values). Each test case can inject about 25~40 messages generated by pairwise algorithm. So invalid messages injected are about 1675~2680 in total 67 test cases. Test verdicts are based on robustness requirement (see section 3.2) and the fail rate is 8/67=11.9%.

Based on test results, critical vulnerabilities of Zebra-0.94 are analyzed: 1) Zebra cannot parse invalid messages with mutated "length" field in OSPF header robustly and the OSPF routine crashes. Analyzing the source code, we find that the checksum routine (in_cksum in checksum.c) does not compare the "length" field in OSPF header with the "length" field in the IP header so that the routine reads past the end of the heap into unauthorized memory space. 2)LSA header also exists the same vulnerability. The LSA checksum routine (ospf_lsa_checksum in ospf_lsa.c) does not verify the validity of the length field in the LSA header. This occurs only for LSA Header in LSU packets.

7 Conclusions and Future Work

It is desirable to test the robustness, reliability of network devices. The work and contribution of this paper are given as follows: we build a novel NPEFSM to effectively guide robustness testing; Normal-Verification Sequence is proposed to enhance verdict mechanism; we apply pairwise strategy to systematically import anomalies to mutate messages and further generate compound test cases which can simplify test sequences; TTCN-3 is extended to describe compound anomalous test cases. While our approach and test system are generic enough to be applied to all protocols, we will focus on application layer protocols (e.g. HTTP, SMTP) in future work as they typically handle human user level inputs that may have more faults. We will also apply our method to test real-time distributed systems.

Acknowledgment. This work is supported by the National Natural Science Foundation of China under Grant No. 60572082.

References

1. National Vulnerability Database, http://nvd.nist.gov/
2. IEEE Standard Glossary of Software Engineering Terminology. IEEE Std 610.12-1990, p. 64 (1990)
3. Pothamsetty, V., Akyol, B.: A Vulnerability Taxonomy for Network Protocols: Corresponding Engineering Best Practice Countermeasures. In: IASTED Internet and Communications conference, US Virgin Islands (November 2004)
4. Fernandez, J.-C., Mounier, L., Pachon, C.: A Model-Based Approach for Robustness Testing. In: The 17th IFIP International Conference on Testing of Communicating Systems (TestCom 2005), Concordia, Canada, May 30-June 2 (2005)
5. Saad-Khorchef, F., Rollet, A., Castanet, R.: A framework and a tool for robustness testing of communicating software. In: ACM Symposium on Applied Computing (SAC 2007), pp. 1461–1466 (2007)
6. Oulu University Secure Programming Group. PROTOS (2002), http://www.ee.oulu.fi/research/ouspg/protos/index.html
7. Xiao, S., Li, S., Wang, X., Deng, L.: ARF, Cisco Systems, Inc. Fault-oriented Software Robustness Assessment for Multicast Protocols. In: Proceedings of the Second IEEE International Symposium on Network Computing and Applications (NCA 2003) (2003)
8. Xiao, S., Deng, L., Li, S., Wang, X.: ARF, Cisco Systems, Inc. Integrated TCP/IP Protocol Software Testing for Vulnerability Detection. In: Proceedings of the 2003 International Conference on Computer Networks and Mobile Computing (ICCNMC 2003) (2003)
9. Turcotte, Y., Tal, O., Knight, S.: Security Vulnerabilities Assessment of the X.509 Protocol by Syntax-Based Testing. In: Military Communications Conference 2004 (MILCOM 2004), Monterey CA, October 2004, vol. 3, pp. 1572–1578 (2004)
10. Tal, O., Knight, S., Dean, T.: Syntax-based Vulnerability Testing of Frame-based Network Protocols. In: Proc. 2nd Annual Conference on Privacy, Security and Trust, Fredericton, Canada, October 2004, pp. 155–160 (2004)
11. Banks, G., Cova, M., Felmetsger, V., Almeroth, K., Kemmerer, R., Vigna, G.: SNOOZE: toward a Stateful NetwOrk prOtocol fuzZEr. In: Information Security Conference (ISC), Samos Island, Greece (September 2006)
12. Neves, N.F., Antunes, J., Correia, M., Veríssimo, P., Neves, R.: Using Attack Injection to Discover New Vulnerabilities. In: Proceedings of the International Conference on Dependable Systems and Networks (DSN 2006), June 2006, pp. 457–466 (2006)
13. FuzzingTools, http://www.scadasec.net/secwiki/FuzzingTools
14. Vasan, A.M.M.: ASPIRE: Automated Systematic Protocol Implementation Robustness Evaluation. In: Proceedings of the 2004 Australian Software Engineering Conference (ASWEC 2004), April 2004, p. 241 (2004)
15. Lei, Y., Tai, K.C.: In-parameter-order: a test generation strategy for pairwise testing. In: Proceedings Third IEEE Intl. High-Assurance Systems Engineering Symp., pp. 254–261 (1998)
16. Tai, K.C., Lei, Y.: A test generation strategy for pairwise testing. IEEE Trans on Software Engineering 28(1), 109–111 (2002)
17. Jing, C., Wang, Z., Shi, X., Yin, X., Wu, J.: Mutation Testing of Protocol Messages Based on Extended TTCN-3. In: Proceedings of the IEEE 22nd International Conference on Advanced Information Networking and Applications (AINA 2008), Japan, pp. 667–674 (2008)
18. Lee, D., Yannakakis, M.: Principles and Methods of Testing Finite-State Machines-A Survey. Proceedings of IEEE 84(8), 1089–1123 (1996)

19. Alur, R., Courcoubetis, C., Yannakakis, M.: Distinguishing tests for nondeterministic and probabilistic machines. In: Symposium on Theory of Computer Science, pp. 363–372. ACM, New York (1995)
20. ETSI: ETSI Standard ES 201 873-1 V3.2.1(2007-03): The Testing and Test Control Notation version 3; Part 1: TTCN-3 Core Language. European Telecommunications Standards Institute (ETSI), Sophia-Antipolis, France (2007)
21. Moy, J.: OSPF Version 2, RFC 2328 (April 1998)
22. Zebra-0.94, http://www.zebra.org/
23. Yin, X., Wang, Z., Wu, J., Jing, C., Shi, X.: Researches on a TTCN-3-based protocol testing system and its extension. Science in China Series F: Information Sciences (accepted, 2008)