The 1st Workshop on Model-Based Verification & Validation

Directed Acyclic Graph Modeling of Security Policies for Firewall Testing

T. Tuglular, Ö. Kaya, and C. A. Müftüoğlu
Department of Computer Engineering,
Izmir Institute of Technology, Turkey
{*tugkantuglular/ozgurkaya/ardamuftuoglu@iyte.edu.tr*}

F. Belli
Department of Computer Science, Electrical Engineering and Mathematics,
University of Paderborn, Germany
{*belli@adt.upb.de*}

Abstract

Currently network security of institutions highly depend on firewalls, which are used to separate untrusted network from trusted one by enforcing security policies. Security policies used in firewalls are ordered set of rules where each rule is represented as a predicate and an action. This paper proposes modeling of firewall rules via directed acyclic graphs (DAG), from which test cases can be automatically generated for firewall testing. The approach proposed follows test case generation algorithm developed for event sequence graphs. Under a local area network setup with the aid of a specifically developed software for this purpose, generated test cases are converted to network test packets, test packets are sent to the firewall under test (FUT), and sent packets are compared with passed packets to determine test result.

***Keywords**: Firewalls, Firewall Policies, Directed Acyclic Graphs, Event Sequence Graphs, Firewall Testing, Security Testing.*

1. Introduction

Firewalls, which act as the most important defense mechanism of network security, have to be tested to validate that they work as specified. The firewall specification is mainly composed of intended security policy and allowed network protocols, which are usually the main focus of an attacker. The intended security policy consists of firewall rules, which configure the firewall behavior, and allowed network protocols. These constitute an important part of firewall's internal infrastructure which can be described as packet capture, decision making on the packet under consideration, and packet release. Decision making operation is carried out with respect to firewall policy and network protocols. The security policy is external to the firewall like a configuration file, whereas packet checking with respect to network protocols is implemented in the firewall software.

Since the firewall policy is considered as a specification and can be represented by a formal model, we propose a model-based testing approach for firewalls. The novelty of this approach is using DAG model for firewall testing. This paper proposes modeling of firewall rules and generating test cases using DAGs. Since event sequence graphs (ESG) are directed graphs, we applied its test case generation algorithm to the DAG representation of firewall rules. Then test packets derived from generated test cases are sent to the firewall to analyze its behavior.

Next section summarizes related work before Section 3 outlines background and the test generation algorithm. The core of the paper, Section 4, presents our firewall testing approach. Sections 5 and 6 include implementation details of the approach and a case study on a firewall. Section 7 concludes the paper and outlines our research work planned.

2. Related Work

A firewall controls network traffic to and from a computer, based on a security policy. Although systematic testing was an omitted area in firewall studies and relative literature, recent studies on firewalls began to fill this gap. Recent work about firewalls mention how firewalls suffer due to design [1] and configuration problems [2, 3, 4]. Including [1] some approaches propose using formal languages to specify firewall rules [5, 6]. In contrast to formal languages defined, the structural testing approach stated in [13], does not consider the rule sequence in firewalls but aims to evaluate firewall by three aspects; namely consistency, completeness, compactness.

In some cases, firewalls are undertaken by points of design, implementation, and configuration, other approaches focus on the analysis of firewalls with anomaly detection by Ehab Al-Shaer and Hazem Hamed in [14]. They also offer an algorithm to discover and detect anomalies. Frantzen, Kerschbaum, Schultz and Fahmi in [9] proposed that given the large number of firewall vulnerabilities that have surfaced in recent years, it is important to develop a comprehensive framework for understanding both what firewalls actually do when they receive incoming traffic and what can go wrong when they process this traffic. They used dataflow model of firewall internals to study vulnerabilities in firewalls.

The studies based on data flow testing [7,8,10,11] have been restricted to testing data dependencies that exist within a procedure which requires information about the flow of data including calls and returns across procedure boundaries. Intra-procedural data flow tests focus on source code by building and searching program's def-use graph and determine the dependencies or definition use pairs. Although existed inter-procedural data flow algorithms cannot provide information about locations of definitions needed for inter-procedural data flow testing, they help in determining the def-use information and guiding selection as well as execution of test cases that meet requirements [12].

Fulp proposes a DAG approach for representing firewall rules in [15], which focuses on the precedence of the rule sequences and reorder rules to improve performance by decreasing packet delay. Also in [16], it is proven that by the approach of DAG the linear sorting of a firewall policy provides an integrity. Our approach follows formal representation of firewall rules with DAG and builds on test case generation algorithm developed for ESG.

3. Background

The firewall policy rules are required to be modeled by a formal specification tool. To represent legal and illegal statements better, a graph-based approach is constituted. In the graph-based representation of a firewall policy, cycles should be avoided. Otherwise, anomalies may occur. The directed acyclic graph representation of firewall rules fulfills these requirements.

3.1 Directed Acyclic Graphs

DAG is a directed graph with no directed circuits. For any vertex v, there is no nonempty directed path that starts and ends on v [19]. The simplest example of a DAG can be given as a directed tree. The vertices of an n-vertex acyclic directed graph G can be labeled with integers from the set {1, 2, ..., n} such that the presence of the edge (i,j) in G implies that i<j where the edge (i,j) is directed from vertex i to vertex j [20].

A partial order is formed by the reach ability relation in a DAG. DAGs are mainly used to model processes where the flow of information moves in a consistent direction [19]. In our case, we use DAG to represent the firewall policy rules as a rule (evaluation) sequence graph, from which test cases are generated using test case generation algorithm developed for event sequence graphs.

3.2 Event Sequence Graphs

The testing process consists of the execution of the SUT with the produced test inputs and the comparison of the real outputs with the expected ones. If the outputs are in compliance with the expected ones, the test is said to have succeeded, else it fails.

Event sequence graph is an event-based formal model, where the inputs and events are merged and assigned to the vertices of an event transition diagram. The arcs visualize the sequence relation of the events [17]. An ESG is a simple albeit powerful formalism for capturing the behavior of interactive systems. The complete set of interactions is captured in terms of a set of ESGs, where each ESG represents a possibly infinite set of event sequences. An event can be a user stimulus or a system response, punctuating different stages of the system activity.

As stated in [18], each edge in the ESG is marked as a legal event pair (EP). A complete event sequence (CES) represents a walk through the ESG by starting at the entry node and ending at the exit node of the ESG. Entry and exit nodes are not events, they represent entry and exit points of an ESG. Faulty (or illegal) event pairs (FEP) are introduced as the edges of the corresponding $\overline{\text{ESG}}$. Moreover, an EP of the ESG can be extended to a faulty, or an illegal, event triple (FETr) by adding a subsequent FEP to this EP. A faulty event sequence (FES) of the length n consists of n-1 events that form a legal ES of length n-2 and of two events at the end that form an FEP.

As stated in [18], faulty CESs (FCESs) can be constructed using FEPs. A FEP that starts at the entry node of the ESG is also a FCES. Furthermore, a FEP is not executable when it does not start at the entry of the ESG. Hence, it is extended by adding suitable prefixes and the resulting sequence becomes a FCES. Each ES that starts at the entry of the ESG and ends at the first symbol of the FEP is prefixed to the FEP and the resulting sequence becomes a FCES.

3.3 Test Case Generation

We use the method described in [18] which uses an ESG and its complement as input and generates a test set that is complete with respect to model-based coverage criterion. There are mainly two objectives for the test case generation procedure. One is the generation of CESs and the other one is to generate FCESs from the complement of ESG that model the system behavior by considering both the desirable and undesirable parts. With the input of a FCES, the SUT is expected to go to a faulty state and raise a related exception handling mechanism. Hence, CESs are used to test the correct behavior, where the FCESs are used to check the exception handling mechanism.

Given an ESG and the corresponding CESG, the test case generation algorithm generates tests that cover both all event pairs in ESG and all faulty event pairs of the CESG. Note that, the sum of the lengths of the generated CESs and FCESs should be minimal to avoid long chain of events. The test case generation algorithm [18] is presented in Figure 1.

```
Input: an ESG with
$n$ := number of the functional units (modules) of the system that fulfill a
well-defined task;
$length$ := required length of the event sequences to be covered;
for unit 1 TO $n$ do
    Generate appropriate $ESG$ and $\overline{ESG}$;
    for $k := 2$ TO $length$ do
        Cover all ESs of length k by means of CESs subject to minimizing
        the number and total length of the CESs
    end
    Cover all FEPs of unit by means of FCESs subject to minimizing the
    total length of the FCESs
end
```

Figure 1. Test case generation algorithm [18]

4. Firewall Testing Approach

The firewall testing approach proposed here is composed of five phases: (1) generating test cases from firewall policy rules, (2) constructing network test packets from generated test cases, (3) sending constructed test packets to FUT, (4) capturing packets that go through the FUT, and (5) comparing sent and captured packets to determine the test result. Proposed firewall testing approach is summarized in Figure 2.

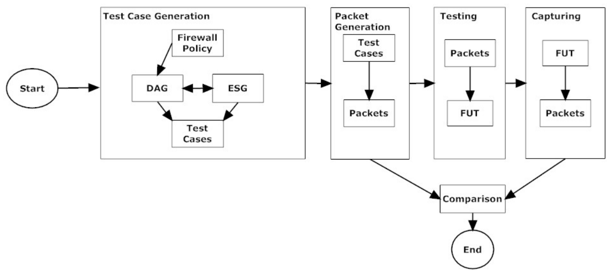


Figure 2. Summary of the approach

The firewall policy parser algorithm explained in [15] is used to convert rules to a DAG and then each rule sequence in the DAG is considered as an event sequence, so that test case generation algorithm for ESG can be utilized. At the test case generation step, the algorithm creates a test case for each complete event (rule) sequence, which are derived from the DAG.

The test case generation algorithm works as follows: for each complete event sequence, a test case is generated from the first rule of that CES and this test case is modified by the proceeding events (rules) in the CES until the "deny all" rule, which is always the last rule for all firewall policies that follows default deny principle. An example of test case generation is given in Section 6.

Once concrete test cases are ready, constructing network test packets as well as sending and capturing them require an appropriate network architecture and some network programming. Our firewall testing tool explained in Section 5 contains the necessary network programming code. To be able to analyze and evaluate the behavior of the firewall under test with respect to test cases, we use an architecture introduced in [23], which is illustrated in Figure 3.

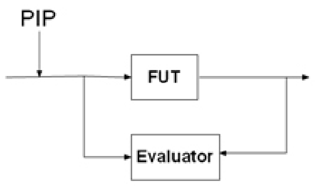
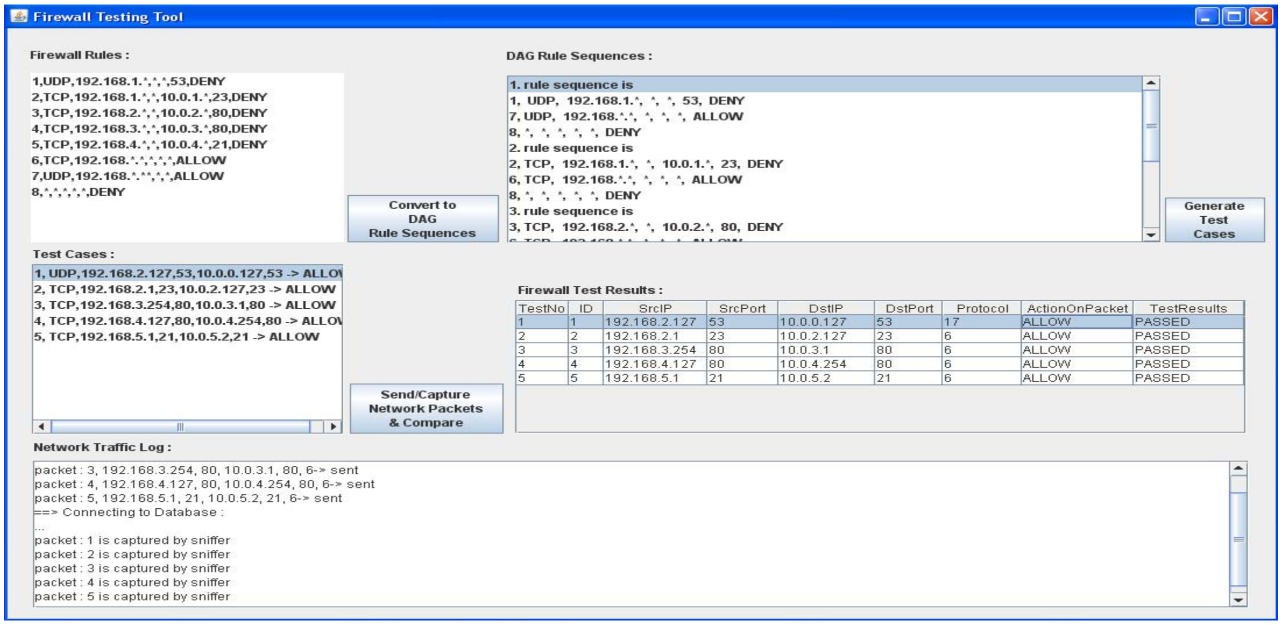


Figure 3. Firewall evaluator architecture [23]

The test packets will be released from packet injection point (PIP), which is the computer that hosts our firewall testing tool. All the traffic entering and leaving the firewall will be recorded and collected data will be analyzed to obtain test outputs, which will be compared with expected outputs to determine test result. We expect to see allowed packets at the packet leaving point but not the denied ones.



```
1,UDP,192.168.1.*,*,*,53,DENY
2,TCP,192.168.1.*,*,10.0.1.*,23,DENY
3,TCP,192.168.2.*,*,10.0.2.*,80,DENY
4,TCP,192.168.3.*,*,10.0.3.*,80,DENY
5,TCP,192.168.4.*,*,10.0.4.*,21,DENY
6,TCP,192.168.*.*,*,*,*,ALLOW
7,UDP,192.168.*.**,*,*,ALLOW
8,*,*,*,*,*,DENY
```

```
1. rule sequence is
1, UDP, 192.168.1.*, *, *, 53, DENY
7, UDP, 192.168.*.*, *, *, *, ALLOW
8, *, *, *, *, *, DENY
2. rule sequence is
2, TCP, 192.168.1.*, *, 10.0.1.*, 23, DENY
6, TCP, 192.168.*.*, *, *, *, ALLOW
8, *, *, *, *, *, DENY
3. rule sequence is
3, TCP, 192.168.2.*, *, 10.0.2.*, 80, DENY
```

```
1, UDP,192.168.2.127,53,10.0.0.127,53 -> ALLO
2, TCP,192.168.2.1,23,10.0.2.127,23 -> ALLOW
3, TCP,192.168.3.254,80,10.0.3.1,80 -> ALLOW
4, TCP,192.168.4.127,80,10.0.4.254,80 -> ALLO
5, TCP,192.168.5.1,21,10.0.5.2,21 -> ALLOW
```

TestNo	ID	SrcIP	SrcPort	DstIP	DstPort	Protocol	ActionOnPacket	TestResults
1	1	192.168.2.127	53	10.0.0.127	53	17	ALLOW	PASSED
2	2	192.168.2.1	23	10.0.2.127	23	6	ALLOW	PASSED
3	3	192.168.3.254	80	10.0.3.1	80	6	ALLOW	PASSED
4	4	192.168.4.127	80	10.0.4.254	80	6	ALLOW	PASSED
5	5	192.168.5.1	21	10.0.5.2	21	6	ALLOW	PASSED

```
packet : 3, 192.168.3.254, 80, 10.0.3.1, 80, 6-> sent
packet : 4, 192.168.4.127, 80, 10.0.4.254, 80, 6-> sent
packet : 5, 192.168.5.1, 21, 10.0.5.2, 21, 6-> sent
==> Connecting to Database :
...
packet : 1 is captured by sniffer
packet : 2 is captured by sniffer
packet : 3 is captured by sniffer
packet : 4 is captured by sniffer
packet : 5 is captured by sniffer
```

Figure 4. Firewall testing tool graphical user interface screenshot

5. Implementation and Tool Support

For the implementation of our approach, we developed a firewall testing tool in Java programming language. The tool reads all rules from a firewall policy and derives test cases from these rules. Then using generated test cases test packets are constructed using JPCAP v0.7 [24] library. Our tool also includes a sniffer to collect packets that pass the FUT.

The generated packets are stored in a database table where MYSQL v5.1 [21] is used as the database management system. The generated packets are sent to the firewall, where IPTABLES v1.4.1.1 [22] is used running under Linux operating system. The packets which pass through the firewall are captured and stored in another table. Both sent and captured packets are compared to determine whether the test is passed or failed. Firewall test results can be seen in the graphical user interface of our tool illustrated in Figure 4.

6. Case Study

The case study was conducted by using an example firewall policy displayed in Table 1. The rules are converted to a DAG and then to an ESG. The ESG model of the example firewall rules is given in Figure 5. It can be seen in the figure that the node r_7 subsumes r_1, r_6 subsumes r_2, r_3, r_4 and r_5 and r_8 subsumes r_6, r_7.

Test generation begins with an analysis of the rules and the subsume relation between them. This analysis leads to the following set of EPs.

(r_1, r_7), (r_2, r_6), (r_3, r_6), (r_4, r_6), (r_5, r_6), (r_6, r_8), (r_7, r_8)

Table 1. Example firewall rules

No	Protocol	Source IP	Source Port	Destination IP	Destination Port	Action
r1	UDP	192.168.1.*	*	*	53	DENY
r2	TCP	192.168.1.*	*	10.0.1.*	23	DENY
r3	TCP	192.168.2.*	*	10.0.2.*	80	DENY
r4	TCP	192.168.3.*	*	10.0.3.*	80	DENY
r5	TCP	192.168.4.*	*	10.0.4.*	21	DENY
r6	TCP	192.168.*.*	*	*	*	ACCEPT
r7	UDP	192.168.*.*	*	*	*	ACCEPT
r8	*	*	*	*	*	DENY

In the next step, CESs are generated. As explained in (Section 3.2), CES is a walk-through obtained by extending the EPs by appropriate suffixes. The list below gives the CESs for rules given in Table 1.

(r_1, r_7, r_8), (r_2, r_6, r_8), (r_3, r_6, r_8), (r_4, r_6, r_8), (r_5, r_6, r_8)

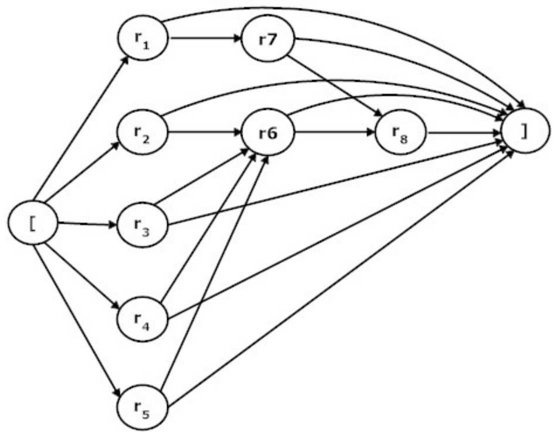


Figure 5. ESG model of the firewall policy showing legal rule sequences

For each complete event (rule) sequence, a test case is generated and then converted to a network test packet. In other words, an individual packet represents a unique test case. For instance, consider the first CES, which is the rule sequence (r_1, r_7, r_8) as shown in Table 2. For this complete sequence, a network test packet is generated with 192.168.2.127 as source IP addresses, with 53 as source port, with 10.0.0.127 as destination IP address, and with 53 as destination port. Table 2 shows generated test packets for each complete event (rule) sequence defined in Table 1 and presents the state of the packets and respectively the test result. In this study, we did not consider FEPs, so FCESs and they are left as future work.

Table 2. Sent and captured packets

No	Test Case	Protocol	Source IP	Source Port	Destination IP	Destination Port	Allowed/Denied	Passed/Failed
1	(r1, r7, r8)	UDP	192.168.2.127	53	10.0.0.127	53	Allowed	Passed
2	(r2, r6, r8)	TCP	192.168.2.1	23	10.0.2.127	23	Allowed	Passed
3	(r3, r6, r8)	TCP	192.168.3.254	80	10.0.3.1	80	Allowed	Passed
4	(r4, r6, r8)	TCP	192.168.4.127	80	10.0.4.254	80	Allowed	Passed
5	(r5, r6, r8)	TCP	192.168.5.1	21	10.0.5.2	21	Allowed	Passed

7. Conclusion

In this paper, we proposed a solution for model-based firewall testing and presented a case study to explain the proposed approach. Using directed acyclic graphs to model firewall rules and utilizing test case generation algorithm of event sequence graphs, we are able to automate firewall testing procedure. This automation is realized with an implemented tool. As future work, we would like to extend our case study by including faulty rule sequences. In the proposed approach, we utilized only concrete test cases, however we would like to integrate the concept of abstract test cases as in [25] to our work. The test case generation utility of the introduced tool can be improved by reflecting hacker methodologies.

References

[1]. Y. Bartal, A. J. Mayer, K. Nissim, and A. Wool, "Firmato: A novel firewall management toolkit", in *Proc. of IEEE Symp. on Security and Privacy*, 1999, pp. 17–31.

[2]. H. Adiseshu, S. Suri, and G. Parulkar, "Detecting and Resolving Packet Filter Conflicts", in *19th Annual Joint Conference of the IEEE Computer and Communications Societies*, 2000, pp. 1203–1212.

[3]. E. S. Al-Shaer and H. H. Hamed, "Discovery of Policy Anomalies in Distributed Firewalls", in *23rd IEEE Computer and Communications Societies Annual Joint Conference*, 2004.

[4]. P. Gupta, "Algorithms for Routing Lookups and Packet Classification", *PhD Thesis*, Computer Science Dept., Stanford University, 2000.

[5]. High level firewall language, *http://www.hlfl.org*, 2009.

[6]. Joshua D. Guttman, "Filtering postures: Local enforcement for global policies", in *Proceedings of IEEE Symp. on Security and Privacy*, 1997, pp. 120-129.

[7]. L.A. Clark, A. Podgurski, D. Richardson, S. Zeil, "A comparison of data flow path selection criteria", in *Proceedings 8th International Conference on Software Engineering*, London, UK, 1985, pp. 244-251.

[8]. B. Korel, J. Laski, "A tool for data flow oriented program testing", in *Proceedings of the Second Conference on Software Development Tools, Techniques, and Alternatives,* 1985, pp. 34-37.

[9]. M. Frantzen, F. Kerschbaum, E. E. Schultz, and S. Fahmy, "A Framework for Understanding Vulnerabilities in Firewalls Using a Dataflow Model of Firewall Internals", in *Proceedings of Computers Security*, 2001, pp. 263-270.

[10]. J. W. Laski, B. Korel. "A data flow oriented program testing strategy", *IEEE Transactions on Software Engineering*, vol. SE-g, no. 3, 1983, pp. 347- 354.

[11]. S. Rapps, E. J. Weyuker, "Selecting software test data using data flow information", *IEEE Transactions of Software Engineering*, vol. SE- 11, no. 4, 1985, pp. 367-375.

[12]. M. J. Harrold, M. L. Soffa, "Interprocedural Data Flow Testing", *SIGSOFT Softw. Eng. Notes,* vol. 14, no. 8, 1989, pp. 158-167.

[13]. M. G. Gouda, A. X. Liu, "Firewall design: consistency, completeness and compactness", in *Proceedings of the 24th IEEE International Conference on Distributed Computing Systems (ICDCS-04)*, 2004, pp. 320-327.

[14]. E. Al-Shaer, H. Hamed, "Discovery of policy anomalies in distributed firewalls", *in IEEE INFOCOM'04*, 2004, pp. 2605-2616.

[15]. E. W. Fulp, "Optimization of Network Firewall Policies using Directed Acyclical Graphs", in *Proceedings of the IEEE Internet Management Conference*, 2005.

[16]. E. W. Fulp, "Firewall policy models using ordered-sets and directed acyclical graphs", *Technical Report*, Wake Forest University Computer Science Department, 2004.

[17]. F. Belli, "Finite-State Testing and Analysis of Graphical User Interfaces", in *Proceedings of the 12th international Symposium on Software Reliability Engineering*, IEEE Computer Society, Washington, DC, 2001, p. 34.

[18]. F. Belli, N. Nissanke, Ch. J. Budnik, A. Mathur, "Test Generation Using Event Sequence Graphs", *Technical Report*, University of Paderborn, 2005.

[19] J. Bang-Jensen, "2.1 Acyclic Digraphs", Digraphs: Theory, Algorithms and Applications, *Springer Monographs in Mathematics*, Springer-Verlag, 2008, pp. 32–34.

[20]. K. Thulasiraman, M. N. S. Swamy, "5.7 Acyclic Directed Graphs", *Graphs: Theory and Algorithms*, John Wiley and Son, 1992, p. 118.

[21]. MySQL, MySQL version 5.1 Community Edition, in *http://dev.mysql.com/downloads/mysql/5.1.html*, 2009.

[22]. IPTABLES, IPTABLES version 1.4.1.1, in *http://www. netfilter.org/projects/iptables*, 2009.

[23]. T. Tuglular, F. Belli, "Model-Based Mutation Testing of Firewalls", *Fast Abstracts of TAIC-PART Conference*, United Kingdom, 2008.

[24]. JPCAP, JPCAP – version 0.7, in *http://netresearch. ics.uci.edu/kfujii/jpcap/doc*, 2009.

[25]. D. Senn, D. Basin, and G. Caronni, "Firewall Conformance Testing", Proc. 17th TestCom, LNCS 3502, Springer, 2005, pp. 226-241.