2011 Fifth International Conference on Secure Software Integration and Reliability Improvement

Mutation-Based Evaluation of Weighted Test Case Selection for Firewall Testing

Tugkan Tuglular
Department of Computer Engineering, Izmir Institute of Technology, Turkey
tugkantuglular@iyte.edu.tr

Gurcan Gercek
Department of Computer Engineering, Izmir Institute of Technology, Turkey
gurcangercek@iyte.edu.tr

***Abstract*—As part of network security testing, an administrator needs to know whether the firewall enforces the security policy as expected or not. In this setting, black-box testing and evaluation methodologies can be helpful. In this paper, we employ a simple mutation operation, namely flipping a bit, to generate mutant firewall policies and use them to evaluate our previously proposed weighted test case selection method for firewall testing. In the previously proposed firewall testing approach, abstract test cases that are automatically generated from firewall decision diagrams are instantiated by selecting test input values from different test data pools for each field of firewall policy. Furthermore, a case study is presented to validate the proposed approach.**

Keywords-testing; firewalls; firewall testing; mutation analysis.

I. INTRODUCTION

Firewalls have to be tested to validate that they work as specified. The main focus of our firewall testing approach is the intended security policy. The intended security policy consists of firewall rules configuring the firewall behavior. The security policy is external to the firewall like a configuration file.

In this paper, firewall policy is represented by firewall decision diagram (FDD) and abstract test cases are generated through path coverage using decision paths on the FDD. Instantiation of abstract test cases with test values, which are selected based on priorities and weights using a feedback control approach, is presented in our previous paper [1]. The test values with high priorities are assumed to have high probability to reveal mismatches between firewall's expected and executed behavior. Weights are used to alternate among high priority test values.

In this paper, we employ a simple mutation operation, namely flipping a bit, to generate mutant firewall policies. Although there are a few studies on mutating specifications and using them to evaluate test sets [2],[3],[4], to the authors' knowledge there is only one study [5] where mutating firewall policies for the evaluation of security test sets is proposed.

The novelty of the approach presented in this paper is to use flipping a bit as a mutation operator to mutate specifications, where we consider firewall policy as a special case of dynamic specifications. The flipping a bit mutation operation is developed specifically for firewall policies. The decision made and action taken by the firewall is either *accept* or *deny*, which can be represented by one bit. A slight change can be obtained by flipping one bit, which means if the action field of a rule in the original policy is *accept*, its corresponding mutant policy will have a rule with *deny* action and vice versa. The flipping a bit mutation operator is a variation of other logic-based mutation operators.

For the generation of mutant policies, we follow table coverage criteria suggested by Ammann and Black [2], whereas Hwang et al. [5] used rule coverage criterion, predicate coverage criterion and clause coverage criterion. The firewall policy is placed in a table, where each rule is a row and each field of a rule is a column. We use mutated policies to evaluate the test set generated using our proposed weighted test selection method.

Next section summarizes related work before Section III outlines the preliminaries required for the proposed approach. Section IV presents our feedback control based test case instantiation approach for firewall testing. The core of the paper, Section V, explains the proposed mutation based evaluation approach for test sets generated for firewall testing. The tool developed for the mutation based evaluation of firewall test sets is presented in Section VI. Section VII presents the case study we carried out to exemplify the proposed approach. Section VIII concludes the paper and outlines future work planned.

II. RELATED WORK

This paper focuses on firewall implementation testing considering only policy execution. There is one approach to firewall implementation testing by Senn et al. [6], who have worked on firewall implementation testing using protocol finite state automata to generate abstract test cases through unique input/output sequences [7] and instantiate abstract test cases with test tuples consisting of

<protocol>, <srcIP>, <dstIP>, <action>

fields of a firewall policy rule. However, in our work abstract test cases are generated from FDD and concrete test cases are built using

<protocol>, <srcIP>, <srcPort>, <dstIP>, <dstPort>, <action>

fields of a firewall rule.

An approach to specification-based test generation for security-critical systems is proposed by Wimmel and Jürjens [8]. Although not directly related, in their work, the test sequences are determined with respect to the security properties required by the system, using mutations of the system specification. They also followed the abstract test case generation approach, however the concretization of abstract test cases apply only to an existing implementation.

Hwang et al. [5] utilized two test case generation method, one is based on local constraint solving and the other one based on global constraint solving, in addition to random test case generation, whereas we employed our weighted test selection method in addition to random test case generation.

In this work, we use FDD [9] notion for modeling, whereas in our previous work [10], we used directed acyclic graph concept to deal with rule dependencies, which is implicitly handled by FDD. The present paper chooses FDD notation since formal, graph-theoretical notions and algorithms are utilized intensively with it.

III. PRELIMINARIES

The firewall testing process starts with the generation of FDD from firewall policy, of which preliminaries are explained in Section III-A. The firewall testing process continues with the generation of abstract test cases and their instantiation with test input values to obtain concrete test cases, of which preliminaries are given in Section III-B. Since the goal of this paper is to use mutation analysis for the evaluation of firewall test sets, the selected test set generation approach for the case study, which is our previously proposed weighted test case selection method [1], is therefore explained in detail in Section IV.

Concrete test cases are converted to network test packets and injected to the firewall under consideration. The packets passing the firewall are collected to determine the result of test cases. The test set evaluation is performed using mutation analysis, of which preliminaries are mentioned in Section III-C and mutation-based evaluation of test sets for firewall testing is explained in detail in Section V.

A. *Firewall Decision Diagram Generation*

While testing firewalls with respect to intended security policy, a model of the firewall policy helps to predict and control its behavior. FDDs are chosen to model the firewall policy.

A firewall decision diagram *f* (or FDD *f*, for short) over the fields $F_0, \cdots, F_{n-1}$ is an acyclic and directed graph [14], where a field F_i is a variable whose value is taken from a predefined interval of nonnegative integers. An FDD *f* over the fields $F_0, \cdots, F_{n-1}$ can be represented by a sequence of rules, each of which is of the form

$F0 \in S0 \wedge \cdots \wedge Fn-1 \in Sn-1 \rightarrow <decision>$

such that the following two conditions hold [9]. First, each rule in the sequence represents a distinct decision path in *f*. Second, each decision path in *f* is represented by a distinct rule in the sequence. Note that the order of the rules in the sequence is immaterial. The algorithm to generate a FDD is presented in [9].

B. *Test Case Generation for Firewalls*

Our test case generation approach consists of two parts. First, we generate abstract test cases. Abstract test cases are produced to test the correct policy handling of a firewall. Second, test input values are collected from various sources such as; firewall policy, domain topology knowledge, and black-lists. To obtain the concrete test cases, we instantiate abstract test cases with test input values.

The sequence of firewall rules is converted to a FDD as described in [9], which is then used for test generation. Each decision path in the FDD represents an abstract test case. We select to abstract a firewall rule as

IF (<protocol>, <srcIP>, <srcPort>, <dstIP>, <dstPort>)
THEN <action>,

where protocol is a network protocol, such as TCP or UDP, and action is either ACCEPT or DENY. The root node of a FDD represents the protocol field, and the terminal nodes represent the action field, intermediate nodes represent other fields respectively. Every decision path starting at the root and ending at a terminal node represents a rule in the policy.

In our previous paper [1], we choose the expert knowledge approach to construct test input values. Although more costly, test input values selected using expert knowledge is assumed to reveal more errors than the other two approaches. Moreover, expert knowledge can prioritize test input values, which is the default feature of our approach. Sets of test input values should be prepared beforehand to be used in the instantiation of the abstract test cases. Although the number of sets may vary from expert to expert, we decide to utilize three sets for each of the following fields: *src_IP*, *dst_IP*, and *dst_port*. We increased the number of the sets proposed in [1]. In this paper, we employed four sets, which are given in Table I.

TABLE I. Sets of Test Input Values

Field	Src_IP	Dst_IP	Dst_port
Set1	blacklist_admin	current_domain_addresses	listening_ports
Set2	blacklist_3rdparty	past_domain_addresses	vulnerable_ports
Set3	past_traffic_addresses	past_traffic_addresses	past_traffic_ports
Set4	policy_addresses	policy_addresses	policy_ports

Finally, we instantiate the abstract test cases with the test input data to obtain concrete test cases. Once the concrete test cases are generated, they are converted to network packets and injected to firewall, where the evaluation is performed using firewall testing architecture as explained in [1].

C. *Mutation-Based Test Set Evaluation*

Mutation testing is a fault-based testing technique providing a mutation adequacy score, which can be used to measure the effectiveness of a test set in terms of its ability to detect faults [11]. The hypothesis behind mutation testing is that the faults introduced by mutation testing represent the mistakes that programmers often make. A mutation operator creates a slight change [12] in the corresponding context, which can be a program, a specification, or a policy in our case. A slightly changed program is called a mutant. To evaluate the quality of a given test set, each mutant is executed against the test set. If the result of running a mutant is different from the result of running the original program for any test cases in the test set, the seeded fault denoted by the mutant is detected [11]. The detection ratio, called mutation score, is used to assess the quality of the test set.

Since most coverage metrics apply to source code, it is difficult to utilize them in cases of conformance testing [2]. Security testing, a kind of conformance testing, is performed with respect to a security policy. Therefore, an approach which evaluates security test sets independent of code is necessary. Ammann and Black [2] developed a specification-based coverage metric to evaluate test sets. We follow their approach and apply it to security policies. Instead of specification mutants, we generate policy mutants using a mutation operator, which we call flipping a bit. Then we evaluate firewall test set for mutation adequacy. The mutation score for a test set is the percentage of non-equivalent mutants killed by that test set. A test set is called mutation adequate if its mutation score is 100% [13].

IV. WEIGHTED TEST CASE SELECTION

For the instantiation of abstract test cases, we proposed feedback control based approach to select test input values [1]. The field values that have higher potential to reveal errors should be selected more often than others. In order to facilitate this idea, priorities should be stored along with field values in the sets of test input values and used in the selection process. Moreover, the sets should have dynamic weights so that alternating among sets is possible. The proposed approach is illustrated in Fig. 1.

The controller is responsible for the determination of next weight vector (wv), which is composed of *n* weight values, where *n* is the total number of sets of test input values. A weight value is a real number and it is initially equal to 1. The controller, remembering the current weight vector and using the feedback, namely the identity (ID) of selected set, determines the next weight vector using Equations (1) and (2).

$$wv_i(k+1) = wv_i(k) - wap \text{ if } i \text{ is the ID of selected set} \quad (1)$$

$$wv_j(k+1) = wv_j(k) + (wap/n-1) \text{ for all } j \text{ not equal to } i \quad (2)$$

where $wv_i(k+1)$ is the the *i*th element of the weight vector at step k+1 and weight alternate percentage (wap) is a real number in (0,1). When there is a test value request, the selector chooses a test input value from the sets using the (setID,elementID) information that comes from the intensity calculator. The intensity calculator stores a priority vector (pv), which is composed of priorities of all elements of all sets.

$$pv = (p_{11}, p_{12},\ldots,p_{1i},p_{21}, p_{22},\ldots,p_{2j},\ldots,p_{n1}, p_{n2},\ldots, p_{nk}) \quad (3)$$

The size of the priority vector is the total number of elements of all sets. For instance, assuming that each set given in Table I has three elements, the size of the priority vector is nine. The intensity vector (iv) is obtained by normalizing the priorities, which is achieved by dividing each priority to the sum of all priorities (Σp).

$$iv = (i_{11}, i_{12},\ldots,i_{1i},i_{21}, i_{22},\ldots,i_{2j},\ldots,i_{n1}, i_{n2},\ldots, i_{nk}) \quad (4)$$

where $i_{jk} = p_{jk} / \Sigma p$.

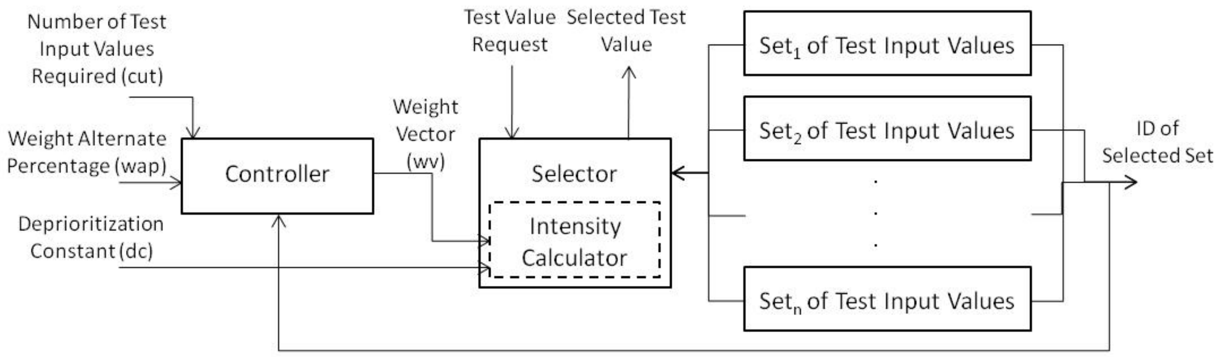

Number of Test Input Values Required (cut)
Weight Alternate Percentage (wap)
Deprioritization Constant (dc)
Controller
Weight Vector (wv)
Test Value Request
Selected Test Value
Selector
Intensity Calculator
Set$_1$ of Test Input Values
Set$_2$ of Test Input Values
Set$_n$ of Test Input Values
ID of Selected Set

Figure 1. Schematic Working of Feedback Control based Approach to Select Test Input Values [1].

The weighted intensity vector (wiv) is calculated by the scalar multiplication of the intensity vector with expanded weight vector (ewv), where expanded weight vector is composed of element weights that have their set weights.

$$wiv = iv \cdot ewv \tag{5}$$

The selected test value is the one where weighted intensity is the maximum of all weighted intensities. The intensity calculator passes the (setID, elementID) information of the maximum weighted intensity to the selector, which returns the corresponding test input value to the requestor.

For illustration purposes, an example is given in Table II. In this example, there are three sets of test input values, each having two elements and their priorities are presented in the priority vector. Using (4) and (5), weighted intensity of each element is calculated and presented in the weighted intensity vector. The element that has the highest weighted intensity is selected as the test value input for that step.

At the next step, the weight of its set is reduced by the controller using *wap*. Additionally, the priority of the selected element is decremented by the intensity calculator using deprioritization constant (*dc*), which is used to lower the priority of an element so that other elements will have a better chance to be selected. The algorithm for feedback control based selection of test input values is given in [1].

The feedback control based selection should be performed separately for each field in the abstract test case. We suggest that test input values for the Protocol, Src_IP, Dest_IP, Dest_port fields should be selected for firewall testing.

V. MUTATION-BASED EVALUATION OF TEST SETS FOR FIREWALL TESTING

A coverage metric independent of implementation is necessary to evaluate test sets generated for firewall testing. In this paper, a mutation-based policy coverage metric is presented analogous to specification coverage metric suggested by Ammann and Black [2]. Although the general idea is similar, our approach differs from theirs in the following points:

- We apply mutations to firewall policies, which are simple logic formulae, whereas they apply mutations to temporal logic formulae, which are more general.
- We use flipping a bit mutation whereas they employed replace constant, replace operator, replace variable, and remove expression mutations.
- We use policy decision point [14], which is an engine that makes accept/deny decisions based on a set of policies, whereas they utilized model checker to execute test cases.

The calculation of mutation-based policy coverage metric for a policy and test set is shown in Fig. 2. First, mutant policies (*MP*s) are generated using a mutation method (*M*) from the original firewall policy (*P*). A generated mutant policy is not accepted as a valid *MP* (*VMP*) if it is same with *P*.

TABLE II. Test Input Value Selection Example using Feedback Control based Selection [1].

		Weight Vector				**Priority Vector**							**Weighted Intensity Vector**									**Selected Set**
Step		w1	w2	w3		A1	A2	B1	B2	C1	C2		wiA1	wiA2	wiB1	wiB2	wiC1	wiC2		**MAX**		
1		1,00	1,00	1,00		5	5	8	4	5	2		0,14	0,14	**0,22**	0,08	0,14	0,05		**0,22**		2
2		1,05	***0,90***	1,05		5	5	*7*	4	5	2		0,15	0,15	0,18	0,08	0,15	0,06		0,18		2

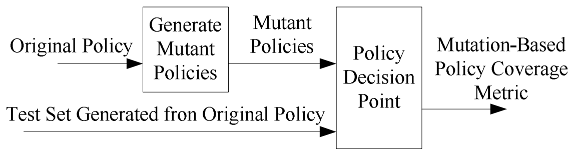

Original Policy
Generate Mutant Policies
Mutant Policies
Policy Decision Point
Mutation-Based Policy Coverage Metric
Test Set Generated fron Original Policy

Figure 2. Calculation of Mutation-Based Policy Coverage.

The test set (*TS*) is executed on *VMP*s at the policy decision point. Mutation-based policy coverage metric is mutation score (*MS*), which is the number of *MP*s killed (*KMP*s) by *TS* divided by the total number of *VMP*s.

$$MS(P, TS, M) = \# \text{ of } KMP\text{s} \,/\, \# \text{ of } VMP\text{s} \qquad (6)$$

The flipping a bit mutation method is developed specifically for firewall policies. The decision made and action taken by the firewall is either *accept* or *deny*, which can be represented by one bit. If the necessary condition explained below occurs, then the slight change can be obtained by flipping one bit, which means if the action field of a rule in the original policy is *accept*, its corresponding mutant policy will have a rule with *deny* action and vice versa.

One of the important questions in mutant generation is when to stop generating mutants. We follow table coverage criteria in our mutant policy generation algorithm. A policy can be represented as a table. The number of rows is equal to the number of rules in the policy and the number of columns is always equal to six. We generate a mutant policy for each cell of the policy table. If we take the policy of the case study as an example, our algorithm will generate 21* 6 = 126 mutant policies.

A mutant policy is generated depending on the value of each cell in the policy table. The function *create_mutant* copies all the rules of the original policy except the current rule and changes the current rule depending on the corresponding cell value.

The value of first cell in the current rule is changed to another protocol and values of the remaining cells are copied. The last argument of *create_mutant* function indicates whether to apply the mutation operator to the action cell or not. A *FALSE* value means “do not apply”, whereas a *TRUE* value means “do apply”.

If the value of second cell or fourth cell in the current rule is a concrete IP address, then a value from complementing address space is selected randomly and mutation operator is not applied to action cell. If the value of second cell or fourth cell in the current rule is an IP address range, then an address is selected randomly from that range and mutation operator is applied to action cell.

If the value of third cell or fifth cell in the current rule is a concrete port value, then a value from complementing port space is selected randomly and mutation operator is not applied to action cell. If the value of third cell or fifth cell in the current rule is a port range, then a value from that port range is selected randomly and mutation operator is applied to action cell.

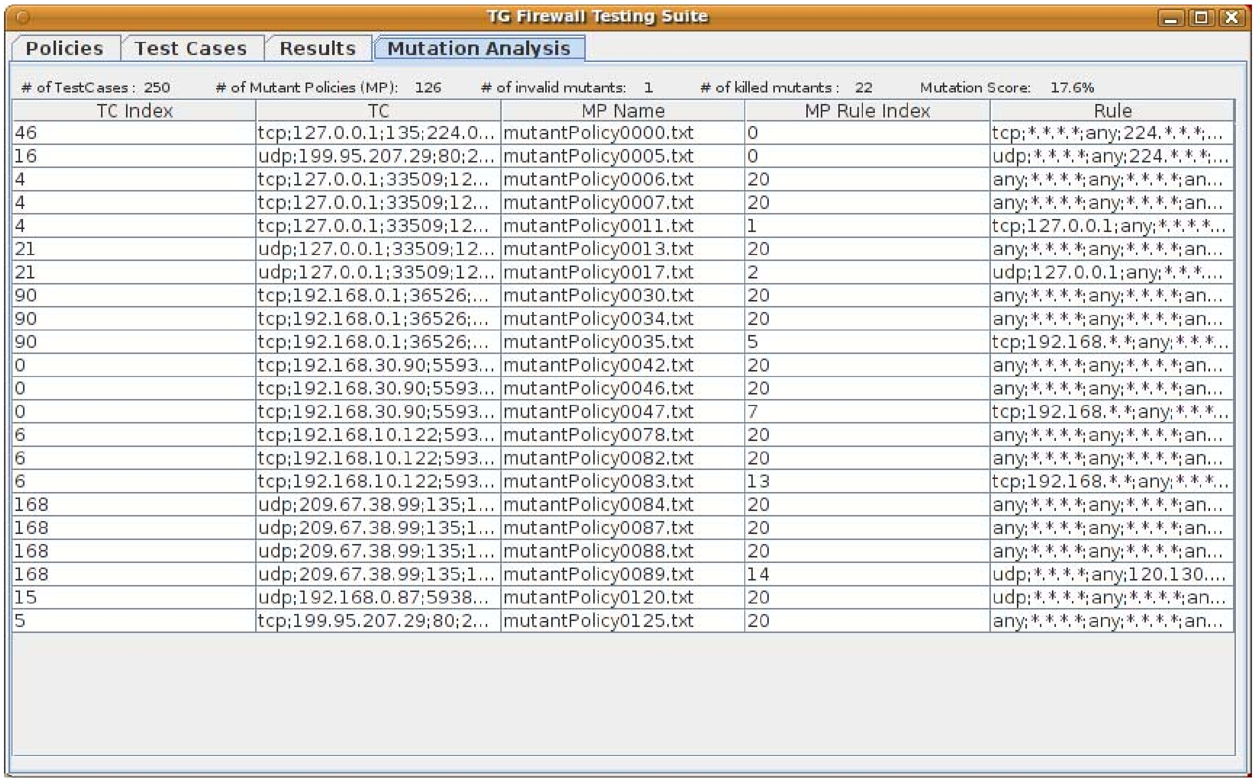
Mutation operator is applied to the sixth cell. After mutant creation for each cell, the resulting mutant policy is compared with the original policy. If they are the same policy, then the created mutant policy becomes an invalid mutant policy to be discarded, thus nil is returned. If they are different, then mutant policy returned from *create_mutant* function.

VI. IMPLEMENTATION AND TOOL SUPPORT

For the implementation of our approach explained in Section IV and V, we developed a mutation analysis tool called TG Firewall Testing Suite (TGFTS) in Java for firewall testing. As a mutation analysis tool working on firewall policies, TGFTS first creates mutant policies from the original firewall policy using only one mutation operation, flipping a bit. Each created mutant has a slight change from the original policy. The *Policies* tabbed pane creates and lists all the mutant policies. The user is able to check the content of each mutant policy.

The *Test Cases* tabbed pane enables users to generate test cases either using the weighted selection method explained in Section IV or randomly. All the parameters are entered in this pane. After test cases are generated, they are listed with field values. The *Results* tabbed pane shows mutant policies versus test cases matrix. Which test case kills which mutant policy can be found out in this matrix.

The last tabbed pane of TGFTS is called *Mutation Analysis* pane and shown in Fig. 3. This pane is used to list killed mutant policies, the test case that killed the mutant and the rule of the mutant policies that the test case fails. As seen from the figure, number of test cases, number of mutant policies, number of invalid mutant policies, number of killed mutants, and calculated mutation score are presented at the top of *Mutation Analysis* tabbed pane.



TG Firewall Testing Suite

Policies | Test Cases | Results | Mutation Analysis

# of TestCases : 250 # of Mutant Policies (MP): 126 # of invalid mutants: 1 # of killed mutants : 22 Mutation Score: 17.6%

TC Index	TC	MP Name	MP Rule Index	Rule
46	tcp;127.0.0.1;135;224.0...	mutantPolicy0000.txt	0	tcp;*.*.*.*;any;224.*.*.*;...
16	udp;199.95.207.29;80;2...	mutantPolicy0005.txt	0	udp;*.*.*.*;any;224.*.*.*;...
4	tcp;127.0.0.1;33509;12...	mutantPolicy0006.txt	20	any;*.*.*.*;any;*.*.*.*;an...
4	tcp;127.0.0.1;33509;12...	mutantPolicy0007.txt	20	any;*.*.*.*;any;*.*.*.*;an...
4	tcp;127.0.0.1;33509;12...	mutantPolicy0011.txt	1	tcp;127.0.0.1;any;*.*.*.*...
21	udp;127.0.0.1;33509;12...	mutantPolicy0013.txt	20	any;*.*.*.*;any;*.*.*.*;an...
21	udp;127.0.0.1;33509;12...	mutantPolicy0017.txt	2	udp;127.0.0.1;any;*.*.*....
90	tcp;192.168.0.1;36526;...	mutantPolicy0030.txt	20	any;*.*.*.*;any;*.*.*.*;an...
90	tcp;192.168.0.1;36526;...	mutantPolicy0034.txt	20	any;*.*.*.*;any;*.*.*.*;an...
90	tcp;192.168.0.1;36526;...	mutantPolicy0035.txt	5	tcp;192.168.*.*;any;*.*.*...
0	tcp;192.168.30.90;5593...	mutantPolicy0042.txt	20	any;*.*.*.*;any;*.*.*.*;an...
0	tcp;192.168.30.90;5593...	mutantPolicy0046.txt	20	any;*.*.*.*;any;*.*.*.*;an...
0	tcp;192.168.30.90;5593...	mutantPolicy0047.txt	7	tcp;192.168.*.*;any;*.*.*...
6	tcp;192.168.10.122;593...	mutantPolicy0078.txt	20	any;*.*.*.*;any;*.*.*.*;an...
6	tcp;192.168.10.122;593...	mutantPolicy0082.txt	20	any;*.*.*.*;any;*.*.*.*;an...
6	tcp;192.168.10.122;593...	mutantPolicy0083.txt	13	tcp;192.168.*.*;any;*.*.*...
168	udp;209.67.38.99;135;1...	mutantPolicy0084.txt	20	any;*.*.*.*;any;*.*.*.*;an...
168	udp;209.67.38.99;135;1...	mutantPolicy0087.txt	20	any;*.*.*.*;any;*.*.*.*;an...
168	udp;209.67.38.99;135;1...	mutantPolicy0088.txt	20	any;*.*.*.*;any;*.*.*.*;an...
168	udp;209.67.38.99;135;1...	mutantPolicy0089.txt	14	udp;*.*.*.*;any;120.130....
15	udp;192.168.0.87;5938...	mutantPolicy0120.txt	20	udp;*.*.*.*;any;*.*.*.*;an...
5	tcp;199.95.207.29;80;2...	mutantPolicy0125.txt	20	any;*.*.*.*;any;*.*.*.*;an...

Figure 3. TG Firewall Testing Suite, Mutation Analysis Pane.

VII. CASE STUDY

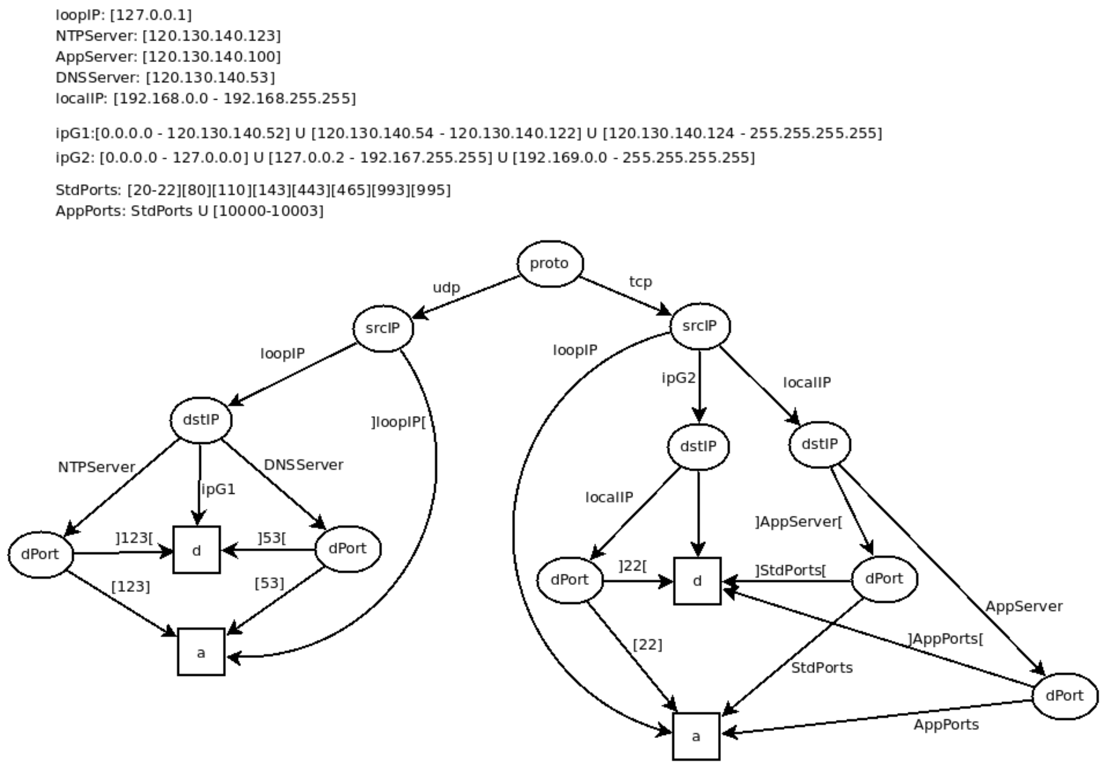
The policy of the firewall is considered as a specification. Therefore, the firewall testing context in this paper is specification based testing, where the operation of FUT or implementation of its policy is checked with respect to its specified policy, i.e. expected behavior. Once the firewall policy is determined, it is loaded to the firewall and the firewall is started. It should be noted that the loaded firewall policy on the FUT can be changed externally after starting the firewall. In that case, firewalls require restart. When that happens, we assume that the specified policy does not match the implemented policy and if there is a mismatch it should be identified. Firewall testing is one of the approaches to identify such a mismatch. Moreover, the firewall software and/or hardware may not behave as expected. The unexpected behavior can also be uncovered by using the firewall testing approach stated in this paper, which is another merit of the proposition.

A. *Firewall Policy Under Consideration*

The firewall policy under consideration (PUC) for the case study is taken from a firewall, which protects a research laboratory in our university. Some of the IP addresses are sanitized for security reasons. Then the policy is converted to a FDD, which is presented in Fig. 4. Using FDD, abstract test cases can be generated by traversing all paths so that path coverage criteria is satisfied. For the right outmost path of the FDD given in Fig. 4, the abstract test case is as follows:

Test Input	Protocol	tcp
Test Input	Src_IP	[192.168.0.0-192.168.255.255]
Test Input	Dst_IP	120.130.140.100
Test Input	Dst_port	20,21,22,80,110,143,443,465, 993,995,[10000-10003]
Expected Output	Action	accept

To instantiate concrete test cases for this abstract test case, test input values should be selected for all fields. The sets of test input values similar to sets given in Table I are determined prior to test input value selection process and employed by our weight based test case selection approach. The test input values for all fields are selected using the proposed weight based selection algorithm with initial-weight=100, wap=0.1 and dc=0.1 values.



loopIP: [127.0.0.1]
NTPServer: [120.130.140.123]
AppServer: [120.130.140.100]
DNSServer: [120.130.140.53]
localIP: [192.168.0.0 - 192.168.255.255]

ipG1:[0.0.0.0 - 120.130.140.52] U [120.130.140.54 - 120.130.140.122] U [120.130.140.124 - 255.255.255.255]
ipG2: [0.0.0.0 - 127.0.0.0] U [127.0.0.2 - 192.167.255.255] U [192.169.0.0 - 255.255.255.255]

StdPorts: [20-22][80][110][143][443][465][993][995]
AppPorts: StdPorts U [10000-10003]

Figure 4. FDD of the Firewall Policy used for Case Study.

For definitive fields, such as Protocol and Dst_IP, the values in the abstract test case are utilized. For Src_port field, which does not occur in the abstract case because it is not in the FDD, a random number generator is employed to generate related test value. After a test case is put together using selected test input values, it is checked for uniqueness. If it exists in the test set, it is discarded and a new test case is formed. After the composition of concrete test cases, network packets are generated from concrete test cases, injected to the network and test results are evaluated.

B. *Results and Discussion*

The PUC for the case study has 21 rules. We generate 126 mutant policies from PUC using mutant policy generation method explained in Section V. One of these mutant policies is consistent with the original PUC, therefore counted as invalid mutant policy.

The test input values for all fields are selected using the weight based selection algorithm with initial-weight=100, wap=0.1 and dc=0.1 values and five test sets, called WTS_i, containing 50, 100, 150, 200, and 250 test cases are generated. Each test set starts with test cases from the preceding test set and adds 50 more test cases to the end. This way, we are able to observe the slope of increase in mutation score. The mutation scores obtained for each weight based selected test set is illustrated in Fig. 5.

Moreover, another five test sets, called RTS_i, containing 50, 100, 150, 200, and 250 test cases are generated randomly. We use the mutation scores of these random generated test sets as a baseline. The mutation scores obtained for each random generated test set is illustrated in Fig. 5 as well.

There are two limitations to be considered with higher number of test cases. One is the limitation introduced by our weighted test case selection approach. The maximum number of test cases to be generated is bounded by the initial weight and the number of elements existing in the sets of test input values.

Second limitation comes from software operational profile modeling research, which shows that after a certain point randomly generated test cases outperforms any other test case generation approach [15]. However, that certain point depends on the operational profile of the software and may be computationally infeasible.

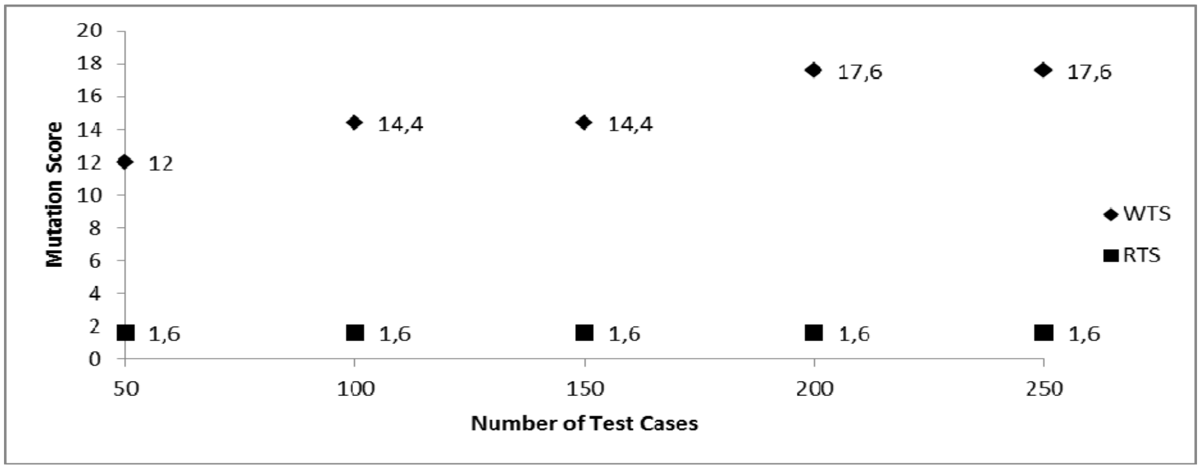


Figure 5. Mutation Scores of WTS and RTS sets.

VIII. CONCLUSION

To evaluate our weighted test case selection and generation approach for firewall testing, we choose mutation analysis method. In the proposed approach, mutant policies are generated from the original firewall policy using flipping a bit mutation. The resulting set of mutant policies are exercised by five test sets, which are generated by our weighted test case selection approach and compared with a different five test sets that are generated randomly. It is observed that for the initial 250 test cases, weighted test case selection and generation approach outperforms random test generation approach for firewall testing.

We will be extending the proposed approach by employing other mutation operators suitable for firewall policies. Then, we would like to compare our weighted test case generation approach with adaptive random testing and intelligent segmentation testing.

REFERENCES

[1] T. Tuglular and G. Gercek, "Feedback Control Based Test Case Instantiation For Firewall Testing", in *7th International Workshop on Software Cybernetics*, July 19, 2010, Seoul, Korea.

[2] P. Ammann and P. Black, "A specification-based coverage metric to evaluate test sets", in *4th IEEE International Symposium on High-Assurance Systems Engineering*, November 17-19, 1999, Washington, D.C.

[3] B. Smith and L. Williams, "Should software testers use mutation analysis to augment a test set?", *Journal of Systems and Software,* vol. 82, pp. 1819-1832, 2009.

[4] A. Gupta and P. Jalote, "An approach for experimentally evaluating effectiveness and efficiency of coverage criteria for software testing", *International Journal on Software Tools for Technology Transfer (STTT),* vol. 10, pp. 145-160, 2008.

[5] H. JeeHyun, X. Tao, C. Fei, and A. X. Liu, "Systematic Structural Testing of Firewall Policies", in *Reliable Distributed Systems, 2008. SRDS '08. IEEE Symposium on*, 2008, pp. 105-114.

[6] D. Senn, D. Basin and G. Caronni, "Firewall conformance testing", *Testing of Communicating Systems,* Springer, pp. 226-241, 2005.

[7] K. Sabnani and A. Dahbura, "A protocol test generation procedure", *Computer Networks and ISDN systems,* vol. 15, pp. 285-297, 1988.

[8] G. Wimmel and J. Jürjens, "Specification-based test generation for security-critical systems using mutations", *Formal Methods and Software Engineering,* pp. 471-482, 2002.

[9] M. Gouda and X. Liu, "Firewall design: Consistency, completeness, and compactness", in *24th International Conference on Distributed Computing Systems*, pp. 320-327, 2004.

[10] T. Tuglular and F. Belli, “Directed Acyclic Graph Modeling of Security Policies for Firewall Testing”, in *1st International Workshop on Model-Based Verification & Validation,* 2009, Shanghai, China.

[11] Y. Jia and M. Harman, "An Analysis and Survey of the Development of Mutation Testing", *IEEE Transactions on Software Engineering,* in print, 2010.

[12] A. Mathur, "Foundations of Software Testing: Fundamental Algorithms and Techniques", *Dorling Kindersley (India) Pvt. Ltd., Pearson Education,* 2008.

[13] A. Offutt, G. Rothermel and C. Zapf, "An experimental evaluation of selective mutation", in *15th International conference on Software Engineering (ICSE '93)*, 1993.

[14] T. Moses, "Extensible access control markup language (xacml) version 2.0", *Oasis Standard,* vol. 200502, 2005.

[15] N. Li, Y. Malaiya, "On input profile selection for software testing", in *5th International Symposium on Software Reliability Engineering*, Nov 6-9, 1994, CA.