A FEEDBACK-BASED TESTING METHODOLOGY FOR NETWORK SECURITY SOFTWARE

A Thesis Submitted to
The Graduate School of Engineering and Sciences of
İzmir Institute of Technology
In Partial Fulfillment of the Requirements for the Degree of

MASTER OF SCIENCE

in Computer Engineering

by
Gürcan GERÇEK

July 2013
İZMİR

We approve the thesis of **Gürcan GERÇEK**

Examining Committee Members:

Assist. Prof. Dr. Selma TEKİR
Department of Computer Engineering, İzmir Institute of Technology

Assist. Prof. Dr. Tuğkan TUĞLULAR
Department of Computer Engineering, İzmir Institute of Technology

Assist. Prof. Dr. Enis KARAARSLAN
Department of Computer Engineering, Muğla Sıtkı Koçman University

11 July 2013

Assist. Prof. Dr. Selma TEKİR
Supervisor, Department of Computer Engineering, İzmir Institute of Technology

Prof. Dr. İ. Sıtkı AYTAÇ
Head of the Department of Computer Engineering

Prof. Dr. R. Tuğrul SENGER
Dean of the Graduate School of Engineering and Sciences

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to my advisors, Assist. Prof. Dr. Tuğkan TUĞLULAR and Assist. Prof. Dr. Selma TEKİR, for their guidance and encouragement. Their enthusiasm, inspiration, and great efforts during this research made this work valuable.

I would also like to state my special thanks to my wife Bilge GERÇEK, for her endless support, motivation and love.

Finally, I would like to express my greatest thanks my parents especially for supporting me throughout my whole life as well as in my graduate study.

ABSTRACT

A FEEDBACK-BASED TESTING METHODOLOGY FOR NETWORK SECURITY SOFTWARE

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 work, 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.

ÖZET

AĞ GÜVENLİĞİ YAZILIMLARI İÇİN GERİ BESLEME TEMELLİ BİR TEST YÖNTEMİ

Ağ güvenlik testlerinin bir parçası olarak, bir yöneticinin, güvenlik duvarının güvenlik politikasını uygulayıp uygulamadığını bilmesi gerekir. Bu çerçevede, kara kutu testi ve değerlendirme metodolojileri faydalı olamaktadır. Bu çalışmada bit değerinin tersini almak olarak bilinen basit mutasyon operasyonlarını, hata barındıran mutant güvenlik duvarı politikaları oluşturması ve bunları daha önceden önerdiğimiz güvenlik duvarı test için ağırlıklandırılmış test durum seçim metodunu değerlendirmesinde kullanıyoruz. Daha önceden önerilmiş güvenlik duvarı testi yaklaşımında, güvenlik duvarı karar diyagramlarından otomatik olarak yaratılan soyut test tanımları, güvenlik duvarı politikasının her bir alanı için farklı test veri kümelerinden seçilmiş test girdi değerleri ile örneklendirilmiştir. Ayrıca önerilen yaklaşımı doğrulamak için bir vaka çalışması sunulmuştur.

TABLE OF CONTENTS

LIST OF FIGURES vii

LIST OF TABLES viii

CHAPTER 1. INTRODUCTION 1

CHAPTER 2. RELATED WORK 6

2.1. Firewall Decision Diagrams 7

2.2. Test Case Generation for Firewalls 12

2.3. Test Architecture for Firewalls 14

2.4. Mutation-Based Test Set Evaluation 15

CHAPTER 3. FEEDBACK-BASED FIREWALL TESTING APPROACH 16

3.1. Abstract Test Case Generation 17

3.2. Weighted Test Case Selection 18

3.3. Test Architecture 21

3.4. Mutation-Based Evaluation of Test Sets for Firewall Testing 23

3.5. Implementation and Tool Support 27

CHAPTER 4. CASE STUDY 29

4.1. Firewall Policy Under Consideration 29

4.2. Results and Discussion 31

CHAPTER 5. CONCLUSION 33

BIBLIOGRAPHY 34

LIST OF FIGURES

<u>Figure</u> **<u>Page</u>**

Figure 1. An FDD 8

Figure 2. Firewall Generation using FDD 8

Figure 3. Marking of FDDs Algorithm 9

Figure 4. Firewall Generation Algorithm 10

Figure 5. Firewall Compaction Algorithm 11

Figure 6. Firewall Simplification Algorithm 12

Figure 7. Firewall Testing Architecture 14

Figure 8. Firewall Testing Process 17

Figure 9. A Simple FDD 18

Figure 10. Schematic Working of Feedback Control based Approach to Select Test Input Values 19

Figure 11. Feedback Control Based Selection of Test Input Values Algorithm 21

Figure 12. Experimental Setup 23

Figure 13. Calculation of Mutation-Based Policy Coverage 24

Figure 14. Mutant Policy Generation Algorithm 25

Figure 15. TG Firewall Testing Suite, Mutation Analysis Pane 28

Figure 16. TG Firewall Testing Suite, Policies Pane 28

Figure 17. FDD of the Firewall Policy Used for Case Study 30

Figure 18. Mutation Scores of WTS and RTS Sets 31

LIST OF TABLES

<u>Table</u> **<u>Page</u>**

Table 1. Sets of Input Values 14

Table 2. Test Input Value Selection Example using Feedback Control based Selection 20

Table 3. Abstract Test Case 30

CHAPTER 1

INTRODUCTION

Computers have an important role in our daily life. From entertainment to health-care they have a vital role to make our lives easier. As a side effect of this, our dependency to computers increases day by day inevitably. Usage of smart phones with social media, email and instant messaging tools allow us contacting with other people anywhere anytime. Other examples from daily lives, our cars can communicate with us by means of navigation systems or for safety measures new trucks can check the distance between the car ahead and itself and activate the breaking system automatically to prevent a crash, if necessary. Examples that are more vital are the security systems, which are responsible to protect systems, data, etc. Companies and Governments try to secure; their networks by deploying firewalls, intrusion detection and/or intrusion prevention systems besides enterprise level authentication systems and anti-virus applications; also using encryption or data loss prevention systems to ensure their privacy and data security.

Computer programs, which are known as software need to be reliable to some certain levels according to its importance. Some inconsistency or aggregation problems about the news in social media may be tolerated or ignored in vast information flows. On the other hand, a malfunctioning in emergency breaking system in trucks/cars cannot be tolerated because of its consequences.

Examples of security software, such as firewalls are expected to have error-free behavior as well. A firewall is a software or hardware-based network security system that controls the incoming and outgoing network traffic by analyzing the data packets and determining whether they should be allowed through or not, based on a rule set.(paraphrase) “The aim of firewall is to protect the network from network-based threats and attacks, and to provide a single choke point where security and audit can be imposed. A firewall builds a blockade between an internal network that is assumed to be secure and trusted, and another network, usually an external (inter)network, such as the Internet, that is not assumed to be secure and trusted.” (Oppliger, 1997). Guarding a trusted environment from untrusted one based on some rule set(can be defined as a firewall policy) is an important task which must not malfunction. Otherwise using a firewall would be the same as using a safe, which can be opened without a key or password.

In order to avoid such consequences software needs to be reliable. To measure the reliability, we need to test software. Software testing is an important topic under Software Engineering discipline. In short we can define software testing as; "an empirical technical investigation conducted to provide stakeholders with information about the quality of the product or service under test." (Kaner, 2006). Testing process tries to detect possible bugs, defects or risks of the usage of the program. It is not feasible to expect to detect all the faults or unexpected behaviors of the program in a single testing approach. In order to increase the detection rates of such undesired effects or behaviors different kind of testing methodologies are suggested, such as regression testing, usability testing, performance testing, security testing, and many more.

Software testing is a part of the quality control and assurance part of software development. Software testing may be defined as the implementation of verification and validation process. According to IEEE Standard Glossary of Software Engineering Terminology (610.12-1990) (Committee, 2013) these terms are defined as follows;

Validation: "The process of evaluating a system or component during or at the end of the development process to determine whether it satisfies specified requirements. Contrast with: ***verification***."

Verification: "(1) The process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase. Contrast with: ***validation***. (2) Formal proof of program correctness. See: ***proof of correctness***."

Verification and Validation (V&V): "The process of determining whether the requirements for a system or component are complete and correct, the products of each development phase fulfill the requirements or conditions imposed by the previous phase, and the final system or component complies with specified requirements. See also: ***independent verification and validation***."

In an informal way to explain, verification is asking the question "Is it really working?" while the validation is asking the question "Is it really working how it is supposed to work?"

Mostly, software’s behaviors do not change during runtime. They behave as programmed and produce output according to given input and parameters. In such cases program flow is static and parameters of the program are used to select the programmed behaviors during the runtime. We can define a program as function f with input set X and parameter set Y. So that output o can be defined as $o = f(x,y), x\ X\ y\ Y$. Some examples are browsers, instant messengers, office products, etc. In such kind of software for each input $x \in X$and parameter $y \in Y$produces same output $o \in O$. On the other hand, some software takes extra arguments to define their behaviors. Such behaviors are not programmed in the software but defined by a given set of rules or expressions, which can be defined as policies. The rules may be defined as some domain specific language or even can be written in any programming language that the software supports, latter method is called as dynamic code execution or dynamic code inclusion. According to this approach we can define this extra argument policy p as $p \in P$ where P is set of policies a software may accept. We need to redefine the function f, $o = f(x,y,p)$so the output o depends also on input p. If we consider p as a set of expressions that will be processed by the program then we can assume that the p is used with some internal parameters z_i of software s and produce some internal output o_i to be used by s, by this assumption we can redefine p as a function; $o_i = p(z_i)$

Firewalls are examples of dynamic behavior software. In this work, we are ignoring the hardware part of the firewalls. The policy parameter of the firewalls may be defined by domain specific languages (DSL) provided by its manufacturer. Enterprise level firewalls provide high level scripting capabilities with different feature sets. Although the representations vary, the basic functionality does not change. Basically a firewall checks the incoming or outgoing packets’ field values against to its policy. If there is a match then the matched rule's action will be applied otherwise default action will be applied to packet under inspection, this will provide completeness to firewall. A policy is composed of rules, basically a rule is a simple boolean expression over packets fields with an action result; <expression> → <action> the action field may contain accept and deny values. A simple rule for blocking all TCP traffic to IP address 192.168.10.43;

$$protocol = TCP \wedge destinationIP = 192.168.10.43 \rightarrow deny$$

Firewall policies can be represented by a simplified five tuple format. For the sake of simplicity we only take into consideration the following five fields over packets.

These fields may change or replace without any problem for the applicability of our approach. These five tuples contain the following fields over packets: Protocol, source IP, source port, destination IP and destination port.

Firewalls have to be tested to validate that they work as specified. Literature on security (Ma, 2004), (Zaugg, 2005) mainly focuses on testing of firewall rules where firewall implementation is assumed error-free. However, a firewall can be hacked and programmed to behave differently from its specification or may have vulnerabilities as shown by Kamara et al. (2003). A firewall vulnerability is defined as an error made during firewall design, implementation, or configuration, that can be exploited to attack the trusted network that the firewall is supposed to protect (Kamara, Fahmy, Schultz, Kerschbaum, & Frantzen, 2003). One of the important goals of security management is identifying and eliminating vulnerabilities.

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 (Tuglular & Belli, 2008).

In this work, firewall policy is represented by firewall decision diagram (FDD). FDD is a graph based representation for firewall policies. By its properties FDD is able to check the completeness and consistency of given firewall policy. FDD is a directed graph with one root node. In FDD, nodes represent fields from five tuple and edges represent range of values of fields. In this setup, terminal nodes represent the action or decision field of the policy. Nodes may have multiple outgoing edges. Union of edges' range values must be equal to input space of source node's field. Each decision path must be started from root node and ended on terminal nodes.

FDD is used to generate abstract test cases through path coverage using decision paths on the FDD. Abstract test cases are definition of possible input sets, in other words they are equivalence classes of input space for a given firewall policy. We instantiate our test cases from each decision path in order to provide more coverage on input space, which will increase the chance to detect faults/bugs in the firewall. In software testing the path coverage method is used on source code to check the all possible paths can be followed during execution. By its nature this method falls in verification category. Our path coverage method is used not on source code but on input space of a firewall policy. By this way our path coverage methodology falls in the validation category. We are trying to validate the behavior of firewall under certain circumstances.

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 (Tuglular & Gercek, 2010). 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. Once the concrete test cases are ready, firewall testing process is executed using firewall evaluator architecture (Tuglular & Belli, 2009).

In this work, 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 (Ammann & Black, 1999), (Smith & Williams, 2009), (Gupta & Jalote, 2008), to the authors' knowledge there is only one study (Hwang, Xie, Chen, & Liu, 2008) where mutating firewall policies for the evaluation of security test sets is proposed.

The novelty of the approach presented in this work 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 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 (1999), whereas Hwang et al. (2008) 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.

CHAPTER 2

RELATED WORK

This work focuses on firewall implementation testing considering only policy execution. There is one approach to firewall implementation testing by Senn et al. (Senn, Basin, & Caronni, 2005), who have worked on firewall implementation testing using protocol finite state automata to generate abstract test cases through unique input/output sequences (Sabnani & Dahbura, 1988) 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 (Wimmel & Jürjens, 2002). 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. (2008) utilized two test case generation methods, 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 (Gouda & Liu, 2004) notion for modeling, whereas in our previous work (Tuglular & Belli, Directed Acyclic Graph Modeling of Security Policies for Firewall Testing, 2009), we used directed acyclic graph concept to deal with rule dependencies, which is implicitly handled by FDD. The present work chooses FDD notation since formal, graph-theoretical notions and algorithms are utilized intensively with it.

2.1. Firewall Decision Diagrams

A field Fi is a variable whose value is taken from a predefined interval of nonnegative integers, called the domain of F_i and denoted by $D(F_i)$. A firewall decision diagram f (or FDD f, for short) over the fields $F_0, \cdot\cdot\cdot, F_{n-1}$ is an acyclic and directed graph that satisfies the following five conditions:

“1. f has exactly one node that has no incoming edges, called the root of f, and has two or more nodes that have no outgoing edges, called the terminal nodes of f.

2. Each nonterminal node v in f is labeled with a field, denoted by F(v), taken from the set of fields $F_0, \cdot\cdot\cdot, F_{n-1}$. Each terminal node v in f is labeled with a decision that is either accept (or “a” for short) or discard (or “d” for short).

3. A directed path from the root to a terminal node in f is called a decision path. No two nodes on a decision path in f have the same label.

4. Each edge e, that is outgoing of a node v in f, is labeled with an integer set I(e), where I(e) is a subset of the domain of field F(v).

5. Let v be any terminal node in f. The set E(v) of all outgoing edges of node v satisfies the following two conditions:

(a) Consistency: For any distinct e_i and e_j in E(v), $I(e_i) \cap I(e_j) = \emptyset$

(b) Completeness: $\cup e \in E(v)$ I(e) = D(F(v)) where $\emptyset$ is the empty set and D(F(v)) is the domain of the field F(v)” (Gouda & Liu, 2004).

An FDD f over the fields $F_0, \cdot\cdot\cdot, F_{n-1}$ can be represented by a sequence of rules, each of them is of the form

$$F_0 \in S_0 \land \cdot\cdot\cdot \land F_{n-1} \in S_{n-1} \rightarrow \text{<decision>}$$

such that the following two conditions hold (Gouda & Liu, 2004). An example FDD is shown in Figure 1. FDD f given in Figure 1 is defined over fields F_0 and F_1. Both of the fields' domains are in the interval of [0-9]. The labels on the edges describe the domain values of its node.

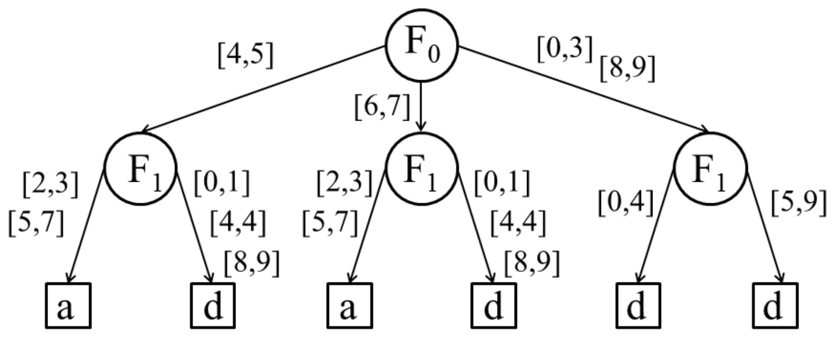


Figure 1. An FDD

These labels must be composed of non-overlapping values. Union of all outgoing edge labels of a node should be equal to domain of that node, otherwise this is a indication of incompleteness of the firewall rules. This condition violates the completeness property of Firewall Decision Diagrams given above.

Each path from root node to terminal nodes can be represented by a rule. From the example this rule can be formed as follows:

$$F_0 \in [4,5] \land F_1 \in [2,3] \cup [5,7] \rightarrow \text{accept}$$

Firewall Decision Diagrams are proposed as method of ideal approach of firewall rule design and implementation.

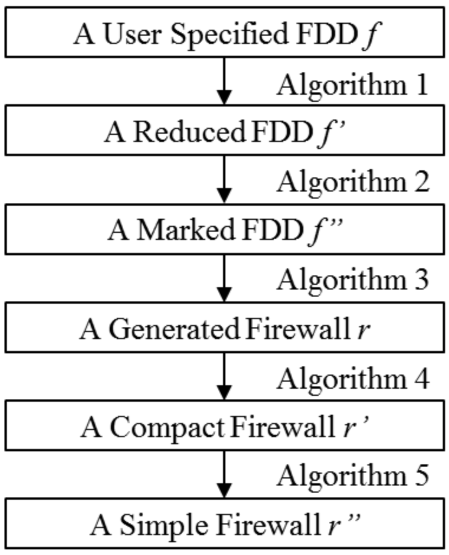


Figure 2. Firewall Generation using FDD

The firewall generation using FDD process is given in Figure 2. In this approach given FDD is reduced by merging the decision paths or edges as much as possible without changing decision paths decision value. After merging, marking process starts. In this process, a marked FDD is generated by marking exactly one outgoing edge of non-terminal nodes in FDD as “ALL”. Any edge marked as “ALL” has a degree of 1 otherwise degree is smallest number of non-overlapping intervals on given edge. This approach leads to many marked version of FDD. The number of rules in the firewall of a marked FDD equals the degree of the marked FDD, thus they tried to minimize the degree by using Algorithm 1 which is shown in Figure 3.

```
Algorithm 1: Marking of FDDs
input : a reduced FDD f
output: a marked version $f'$ of $f$ such that for every marked version $f''$ of $f$, $deg(f') \leq deg(f'')$
steps:
        Compute the degree of each terminal node $v$ in $f$ as follows:
                $deg(v) = 1$

        while $f$ has a node $v$ whose degree has not yet been computed and $v$ has $k$ outgoing edges
        $e_0, \cdots, e_{k-1}$ that are incoming of the nodes $v_0, \cdots, v_{k-1}$, respectively, whose degrees have
        already been computed do
                Find an outgoing edge $e_j$ of $v$ whose quantity
                        $(deg(e_j) - 1) \times deg(v_j)$ is larger than or equal to the corresponding quantity
                        of every other outgoing edge of $v$.
                Mark edge $e_j$ with “ALL”.
                Compute the degree of $v$ as follows:

                        $\deg(v) = \sum_{i=0}^{k-1}(\deg(e_i) \times \deg(v_i))$

        end
```

Figure 3. Marking of FDDs Algorithm

“In Algorithm 3, for generating a firewall of a marked FDD f, which is generated by Algorithm 2. The generated firewall is a sequence of rules where each rule corresponds to a decision path in the marked FDD f . Algorithm 3 computes for each rule in the generated firewall a binary number, called rank of the rule, and two predicates, called exhibited and original predicates of the rule. The rule ranks are used to order the computed rules in the generated firewall. The exhibited and original predicates of the rules are used in the next section to make the generated firewall “compact”.” (Gouda & Liu, 2004). Algorithm 2 is shown in Figure 4 and Algorithm 3 is shown in Figure 5.

```
Algorithm 2: Firewall Generation
input: a marked FDD f over the fields $F_0 , \cdots , F_{n-1}$ and assume that along each directed path in f, if a
field $F_i$ appears before field $F_j$ then $i < j$.
output: a firewall r over the same fields such that
        $r.accept = f.accept$, and
        $r.accept = f.accept$
and for each rule $r_i$ in r, the algorithm computes a binary number of n bits, called the rank of $r_i$, and two
predicates, called the exhibited and original predicates of $r_i$.
steps:
    For each decision path in f, compute a rule $r_i$, its rank, its exhibited predicate $ep_i$ and its original
    predicate $op_i$ as follows:
        $F_0 \in S_0 \wedge \cdots \wedge F_{n-1} \in S_{n-1} \rightarrow decision$
        rank = $b_0 \cdots b_{n-1}$
        $ep_i = (F_0 \in S_0 \wedge \cdots \wedge F_{n-1} \in S_{n-1})$
        $op_i = (F_0 \in T_0 \wedge \cdots \wedge F_{n-1} \in T_{n-1})$
    where each $b_i$, $S_i$, and $T_i$ is computed according to the following three cases:
        Case 1: (The decision path has no nodes labelled $F_i$)
            $b_i = 0$
            $S_i$ =the domain $[a_i , b_i]$ of $F_i$
            $T_i$ =the domain $[a_i , b_i]$ of $F_i$
        Case 2: (The decision path has a node labelled Fi, and its outgoing edge e has no mark)
            $b_i = 0$;
            $S_i$ =the integer set that labels e
            $T_i$ =the integer set that labels e
        Case 3: (The decision path has a node labelled Fi, and its outgoing edge e has an ALL mark)
            $b_i = 1$
            $S_i$ =ALL
            $T_i$ =the integer set that labels e
    After computing all the rules and their ranks, order the rules in r in an ascending order of their
    ranks.
```

Figure 4. Firewall Generation Algorithm

A generated firewall :

$r = (F_0 \in [4, 7] \wedge F_1 \in [2, 3] \cup [5, 7] \rightarrow a,$

$F_0 \in [4, 7] \wedge F_1 \in ALL \rightarrow d,$

$F_0 \in ALL \wedge F_1 \in [0, 9] \rightarrow d)$

“Firewalls that are generated by Algorithm 3 in the last section can have redundant rules, i.e., rules that can be removed from their firewalls without affecting the accept or discard sets of these firewalls.” (Gouda & Liu, 2004). With Algorithm 4 they removed all redundant rules.

```
Algorithm 3: Firewall Compaction
input: a firewall $r$ with m rules ($r_0, \cdots, r_{m-1}$) over the fields $F_0, \cdots, F_{n-1}$ generated by Algorithm 2
output: a compact firewall $r$ such that
        $r'.accept = r.accept$, and
        $r'.accept = r.accept$
variables
        $i$ : $0..m - 2$;
        $j$ : $0..m$;
        redundant : array[$0..m - 1$] of boolean;
        np : name of a predicate;
steps:
        redundant[$m - 1$] := false;
        for $i = m - 2$ to 0 do
            $j := i + 1$;
            let $r_i.op$ be named np;
            redundant[$i$] := true
            while redundant[$i$] $\wedge j \leq m - 1$ do
                if redundant[$j$]
                then $j := j + 1$;
                else if (decision of $r_i$ = decision of $r_j$)
                    $\vee$ (no packet over the fields $F_0, \cdots, F_{n-1}$ satisfies $np \wedge r_j.ep$)
                then let $np \wedge \neg r_j.ep$ be named np;
                    $j := j + 1$;
                else redundant[$i$] := false;
        Remove from $r$ every rule $r_i$ where redundant[$i$] := true;
```

Figure 5. Firewall Compaction Algorithm

Compact firewall example:

$$r = (\quad F_0 \in [4, 7] \wedge F_1 \in [2, 3] \cup [5, 7] \rightarrow a,$$
$$F_0 \in ALL \wedge F_1 \in [0, 9] \rightarrow d)$$

For the last step firewall needs to be simplified. In paper (Gouda & Liu, 2004) simplification process is defined as follows;

“A firewall rule of the form

$$F_0 \in S_0 \wedge \cdot \cdot \cdot \wedge F_{n-1} \in S_{n-1} \rightarrow decision$$

is called simple if every Si in the rule is either the ALL mark or an interval of consecutive nonnegative integers. A firewall is called simple off all its rules are simple. The following algorithm can be used to simplify any firewall generated by Algorithm 3 or Algorithm 4.” (Gouda & Liu, 2004). Algorithm 4 is shown in Figure 6.

```
Algorithm 4: Firewall Simplification
input: a firewall r generated by Algorithm 2 or Algorithm 3
output: a simple firewall r′ such that
        r′ .accept = r.accept, and
        r′ .accept = r.accept
steps:
        while r has a rule of the form
                $F_0 \in S_0 \wedge \cdots \wedge F_i \in S \cup [a, b] \wedge \cdots \wedge F_{n-1} \in S_{n-1} \rightarrow decision$
                where $S$ is a nonempty set of nonnegative integers that has neither $a - 1$ nor $b + 1$
        do
                replace this rule by two consecutive rules of the form:
                $F_0 \in S_0 \wedge \cdots \wedge F_i \in S \wedge \cdots \wedge F_{n-1} \in S_{n-1} \rightarrow decision$ ,
                $F_0 \in S_0 \wedge \cdots \wedge F_i \in [a, b] \wedge \cdots \wedge F_{n-1} \in S_{n-1} \rightarrow decision$
        end
```

Figure 6. Firewall Simplification Algorithm

A simple firewall:

$r = (F_0 \in [4, 7] \wedge F_1 \in [2, 3] \rightarrow a,$

$F_0 \in [4, 7] \wedge F_1 \in [5, 7] \rightarrow d,$

$F_0 \in ALL \wedge F_1 \in [0, 9] \rightarrow d)$

"Our contribution in this paper is two-fold. First, we proposed to use firewall decision diagrams to specify firewalls at the early stage of firewall design. The main advantages of these diagrams are that their consistency and completeness can be checked systematically. Second, we developed a sequence of five algorithms that can be applied to a firewall decision diagram to generate a compact sequence of firewall rules while maintaining the consistency and completeness of the original firewall diagram." (Gouda & Liu, 2004).

In our work we use FDD for both representing the given firewall policies and simplifying them.

2.2. Test Case Generation for Firewalls

Our test case generation 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 (Gupta & Jalote, 2008), 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 in order. Every decision path starting at the root and ending at a terminal node represents a rule in the policy and vice versa.

Sets of test input values may be constructed using equivalence class partitioning, intelligent segmentation (Senn, Basin, & Caronni, 2005), or expert knowledge. The equivalence class partitioning divides the input domain of policy field into a finite number of partitions or equivalence classes (Sabnani & Dahbura, 1988). El-Atawy et al. (2005) proposed intelligent segmentation, where potential erroneous regions in the firewall input space are adapted using the firewall policy. When determining test input data, values that a hacker might choose may be considered in addition to using the blacklists from network/security administrator or third parties as well as using statistical significant/insignificant past traffic. In this work, we choose the expert knowledge approach to construct test input values. Although more time consuming and costly, test input values selected using expert knowledge is assumed to reveal more errors than other two approaches. Moreover, expert knowledge can prioritize test input values, which is the default feature of our approach. Finally, we instantiate the abstract test cases with the test input data to obtain concrete 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 (Tuglular & Gercek, 2010). In this work, we employed four sets, which are given in Table 1.

Table 1. Sets of Test Input Values

Field	Src_IP	Dst_IP	Dst_Port
Set1	Blacklist Admin	Current Domain Addresses	Listening Ports
Set2	Blacklist 3rd Party	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 Section 3.3.

2.3. Test Architecture for Firewalls

Although firewalls are software implementations, the method used for input and output is network I/O. Thus, network packets should be produced, injected, and collected for testing a firewall. Test packets are derived from generated test cases and those packets are sent or injected to the firewall to analyze its behavior. In order to analyze and evaluate the behavior of firewall under test (FUT) with respect to test cases, a special architecture as illustrated in Figure 7 was developed in our previous work (Tuglular & Belli, 2009).

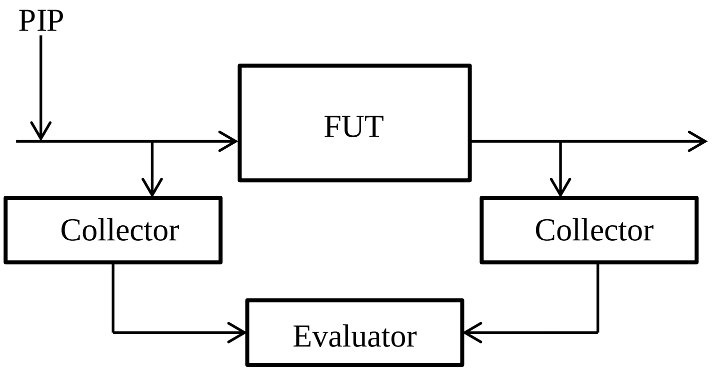


Figure 7. Firewall Testing Architecture

The packet injection point (PIP) is used to release test packets. All the traffic entering and leaving the firewall is recorded and collected data is analyzed to obtain test outputs. The outputs are then compared with expected outputs to determine test result. The accepted packets are expected to be at the packet leaving point, but not the denied packets. The presented architecture can also be used to monitor a firewall constantly in order to check whether it operates in accordance with its specification and implementation (Tuglular & Belli, 2009).

2.4. 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 (Jia & Harman, 2010). 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 (Mathur, 2008) 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 (Jia & Harman, 2010). 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 (Ammann & Black, 1999). 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 (1999) 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% (Offutt, Rothermel, & Zapf, 1993).

CHAPTER 3

FEEDBACK-BASED FIREWALL TESTING APPROACH

The firewall testing process we employed is shown in Figure 8. The process starts with generation of FDD from firewall policy based on the algorithms we mentioned in Chapter 2. Next step covers the generation of abstract test cases. Abstract test cases define sets of test cases. These abstract test cases are generated by traversing paths of generated FDD. All paths must start from root node and end at a terminal node. Each valid path on FDD corresponds to an abstract test case. After the generation of abstract test cases, concrete test cases are instantiated. Instantiation is basically done by selecting an element from the set of test cases defined by an abstract test case. Abstract test case generation will be discussed in Section 3.1.

The selection procedure is based on a simple weighted feedback mechanism. For deciding the weights we used expert knowledge besides the history information of the network, which holds the firewall under consideration. The weighted test case selection method (Tuglular & Gercek, 2010) we used to instantiate concrete test cases is explained in detail in Section 3.2.

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. After the collection of packets, we compare the test cases' expected firewall decision with firewall's actually decision in order to name the test case as passed or failed. Another approach is using simulator rather than real firewall software/hardware. A firewall simulator will mimic the firewall behavior and makes it easier to setup experiments and collect the packets passing on it. The architecture used for testing is explained in Section 3.3.

To measure the quality of test set we used mutation analysis. Mutation analysis is basically generating new policies by using policy under test and use test set against generated policies. The difference between original policy and the generated policy is that generated one contains simple faults in it. Such faults must cause change in behavior of policy otherwise they are not valid. The quality of a test set is measured by the detection rate of changed behavior of generated policies. Mutation analysis will be discussed in Section 3.4.

In Section 3.5 we will explain the tool we developed. The tool is capable of generating both abstract and concrete test cases, evaluation of test cases and mutation analysis as well.

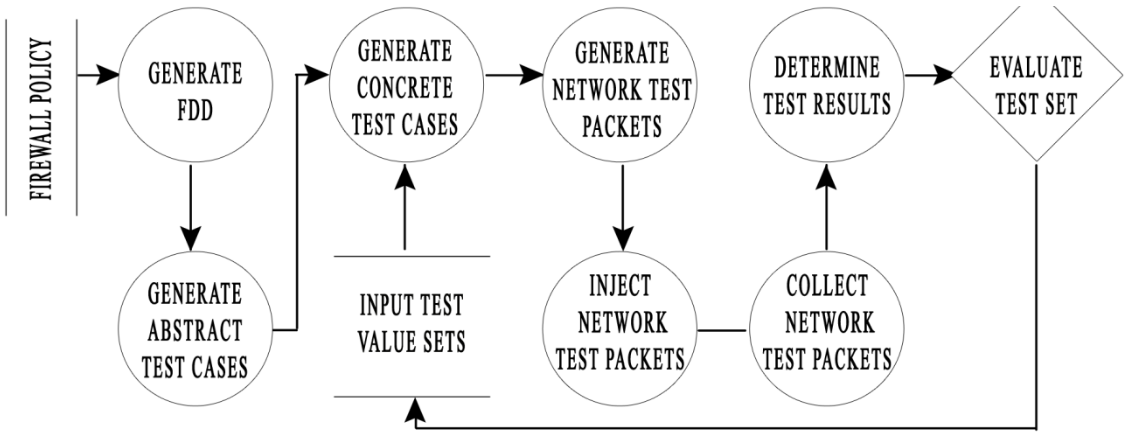

FIREWALL POLICY
GENERATE FDD
GENERATE ABSTRACT TEST CASES
GENERATE CONCRETE TEST CASES
INPUT TEST VALUE SETS
GENERATE NETWORK TEST PACKETS
INJECT NETWORK TEST PACKETS
COLLECT NETWORK TEST PACKETS
DETERMINE TEST RESULTS
EVALUATE TEST SET

Figure 8. Firewall Testing Process

3.1. Abstract Test Case Generation

An abstract test case is a definition of a set of test cases. An abstract test case is generated by selecting a path in a given FDD. A valid path must start at root node and end at one of the terminal nodes. In Figure 9 a simple FDD is given. Let’s follow the right-most path of the FDD in Figure 9. The path is as follows;

Protocol ∈ {TCP}

Source IP ∈ {[1.1.1.1-193.139.255.255] ∪ [193.141.1.1-255.255.255.255]}

Source Port ∈ {[1-65535]}

Destination IP ∈ {140.130.120.80}

Destination Port ∈ {[1-79] ∪ [81-65535]}

Action → Deny

Any element belongs to this set/definition will generate a result of deny at the firewall and this definition is called an abstract test case of FDD in Figure 9. For each path we repeat this procedure to split firewall test input space by it decisions.

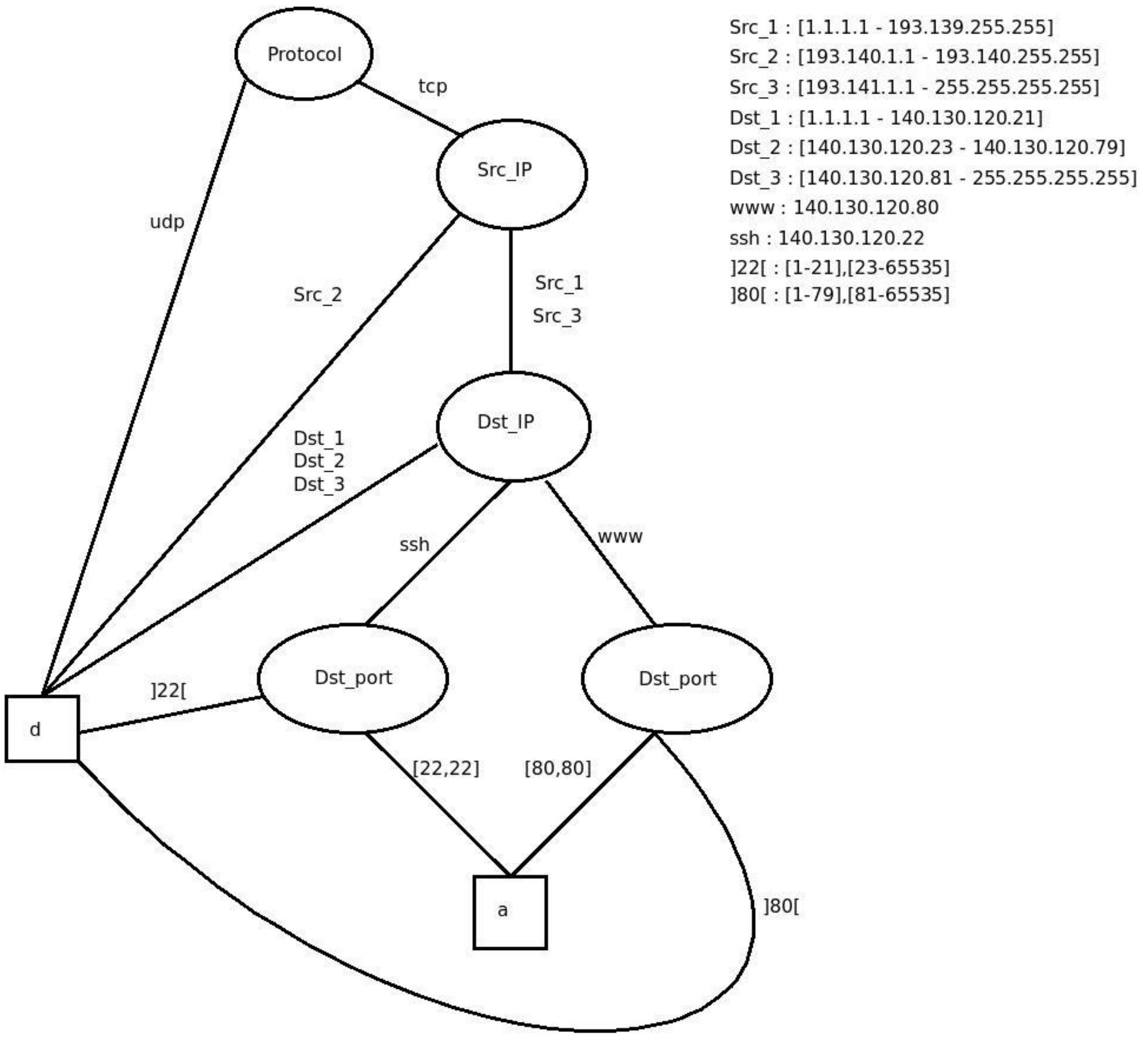

Src_1 : [1.1.1.1 - 193.139.255.255]
Src_2 : [193.140.1.1 - 193.140.255.255]
Src_3 : [193.141.1.1 - 255.255.255.255]
Dst_1 : [1.1.1.1 - 140.130.120.21]
Dst_2 : [140.130.120.23 - 140.130.120.79]
Dst_3 : [140.130.120.81 - 255.255.255.255]
www : 140.130.120.80
ssh : 140.130.120.22
]22[: [1-21],[23-65535]
]80[: [1-79],[81-65535]
Protocol
tcp
udp
Src_IP
Src_2
Src_1
Src_3
Dst_IP
Dst_1
Dst_2
Dst_3
ssh
www
Dst_port
Dst_port
]22[
d
[22,22]
[80,80]
a
]80[

Figure 9. A Simple FDD

3.2. Weighted Test Case Selection

For the instantiation of abstract test cases, we proposed feedback control based approach to select test input values (Tuglular & Gercek, 2010). 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 Figure 10.

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 (3.1) and (3.2).

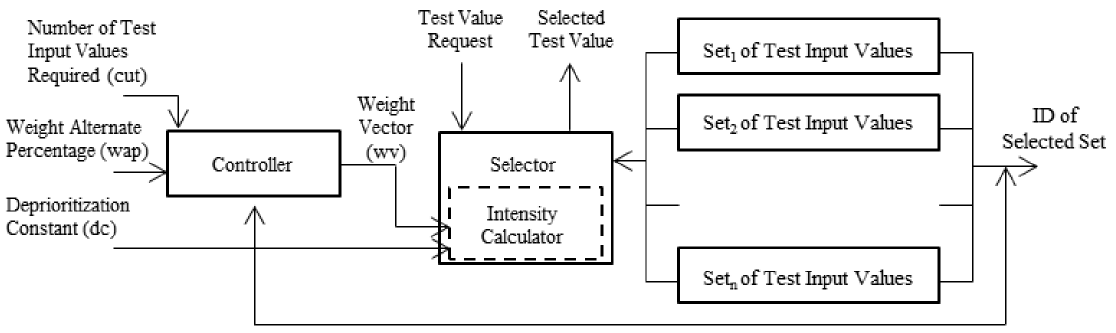

Number of Test Input Values Required (cut)
Weight Alternate Percentage (wap)
Deprioritization Constant (dc)
Controller
Weight Vector (wv)
Test Value Request
Selected Test Value
Selector
Intensity Calculator
Set$_1$ of Test Input Values
Set$_2$ of Test Input Values
Set$_n$ of Test Input Values
ID of Selected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

Figure 10. Schematic Working of Feedback Control based Approach to Select Test Input Values

wvi(k+1) = wvi(k) - wap if i is the ID of selected set (3.1)

wvj(k+1) = wvj(k) + (wap/n-1) for all j not equal to i (3.2)

where wvi(k+1) is the the ith 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 = (p11, p12,...,p1i,p21, p22,...,p2j,...,pn1, pn2,..., pnk) (3.3)

The size of the priority vector is the total number of elements of all sets. For instance, assuming that each set given in Table 1 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 = (i11, i12,...,i1i,i21, i22,...,i2j,...,in1, in2,..., ink) (3.4)

where ijk = pjk / Σp.

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 • ewv (3.5)

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 2. 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 (3.4) and (3.5), 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.

Table 2. Test Input Value Selection Example using Feedback Control based Selection

		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

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 (Tuglular & Gercek, 2010).

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. In the following section, we present a case study to illustrate the application of our proposed approach for feedback control based selection of test input values to firewall testing.

```
Algorithm 5:. Feedback Control based Selection of Test Input Values
input: cut, wap, dc, ev, pv
output: osse
variables:
        cut: number of test input values required
        wap : weight alternate percentage
        dc : deprioritization constant
        ev : element vector
        pv : priority vector
        osse: ordered set of selected elements
steps:
        proceed := true
        while proceed do
                calculate wv using wap
                ewv := extend(wv)
                calculate iv using dc
                wiv := iv • ewv
                selected_set := selectS(ev, max(wiv))
                selected_element := selectE(ev, max(wiv))
                osse := osse ∪ selected_element
                cut := cut – 1
                if (cut == 0 or pv is all zero) then proceed := false

        do while
        return osse
```

Figure 11. Feedback Control Based Selection of Test Input Values Algorithm

3.3. Test Architecture

In Figure 7 we introduced our testing architecture. This method could be called injection based firewall testing. We inject the generated packets on one side of the firewall and collect passed packets on the other side of the firewall, then evaluates the expected results and actual results to see if there is something wrong with the policy enforcement or not.

A real experimental setup is introduced in Figure 12. The components and their responsibilities are as follows.

- *Packet Generator*: This component is responsible for parsing the firewall policy and generate correspondent FDD. Use generated FDD to generate abstract test cases end instantiate concrete test cases. Concreted test cases are passed to sender for injection to network.
- *Sender*: This component is a simple Python script based on Scapy. It took a string representation (5 tuple) of a packet that needs to be generated. According to this representation sender generates a new packet then inject it to network. All activity is logged in a MySQL database, for later evaluation. Another

responsibility is all generated packets are marked. The mark operation is done by generating a unique signature/id for generated packet and put it on the payload. So that on Listener component, we are able to check whether the captured packet is generated by our system or not.

- *Listener*: This component is a simple Python script based on Scapy. It sniffs the outgoing traffic of firewall and records marked packets. All captured marked packets are stored in our database as accepted. In this approach to check the dropped packets, we need to check the logs of firewall whether the marked packets generated by sender but not captured by listener are dropped because of the policy or because of some other problem on firewall or network. This log checking procedure is not generic because of the product specific log formats.
- *Firewall*: We used a PC with two Ethernet interfaces Linux 2.6 installed on it. As firewall software we used iptables, which is easy to configure and easy to collect logs of it. An example on how collect logs on iptables is as follows;

  Log Incoming Dropped Packets:
  ```
  iptables -N LOGGING
  iptables -A INPUT -j LOGGING
  iptables -A LOGGING -j LOG --log-prefix "[Incoming Dropped]: " --log-level 7
  iptables -A LOGGING -j DROP
  ```
  Log Outgoing Droped Packets:
  ```
  iptables -N LOGGING
  iptables -A OUTPUT -j LOGGING
  iptables -A LOGGING -j LOG --log-prefix "[Outgoing Dropped]: " --log-level 7
  iptables -A LOGGING -j DROP
  ```
- *Evaluator*: This component is the decision point of the test. This component connects to MySQL database to compare the generated packets with captured ones. This comparison evaluates only accepted packets, in order to evaluate dropped packets we need to check iptables logs in syslog. By merging results of two comparison method gives us the difference between expected results of generated packets with actual results of firewall decision.

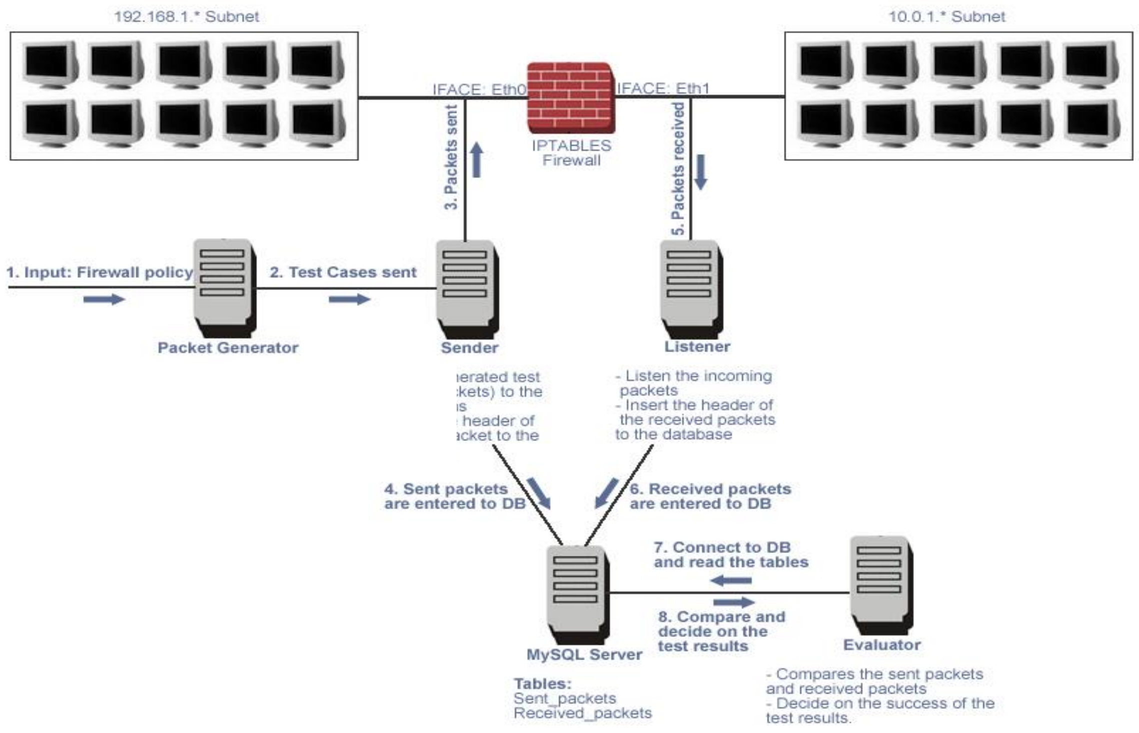

192.168.1.* Subnet
10.0.1.* Subnet
IFACE: Eth0
IFACE: Eth1
IPTABLES Firewall
3. Packets sent
5. Packets received
1. Input: Firewall policy
2. Test Cases sent
Packet Generator
Sender
Listener
- Listen the incoming packets
- Insert the header of the received packets to the database
4. Sent packets are entered to DB
6. Received packets are entered to DB
7. Connect to DB and read the tables
8. Compare and decide on the test results
MySQL Server
Evaluator
Tables: Sent_packets Received_packets
- Compares the sent packets and received packets
- Decide on the success of the test results.

Figure 12. Experimental Setup

Another testing approach is simulation unlike the injection based firewall testing. It is easier and more controlled environment. We don’t need to check the custom log files to see the dropped packets are really dropped because of the policy or some other problem. We developed a simple firewall engine to simulate a firewall based on five tuples.

3.4. 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 work, a mutation-based policy coverage metric is presented analogous to specification coverage metric suggested by Ammann and Black (1999). 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 (Moses, 2005), 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 Figure. 13. First, mutant policies (MPs) 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.

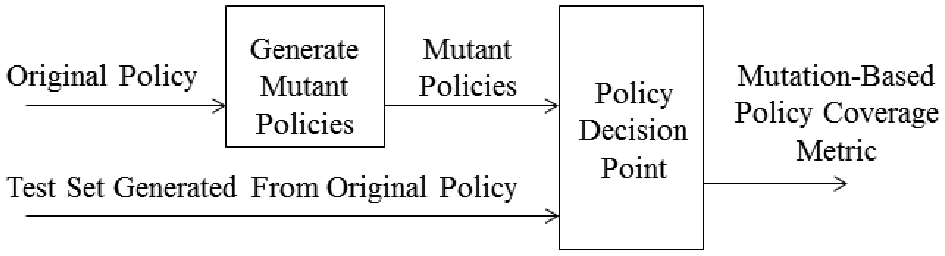

Original Policy
Generate Mutant Policies
Mutant Policies
Policy Decision Point
Mutation-Based Policy Coverage Metric
Test Set Generated From Original Policy

Figure 13. Calculation of Mutation-Based Policy Coverage

The test set (TS) is executed on VMPs at the policy decision point. Mutation-based policy coverage metric is mutation score (MS), which is the number of MPs killed (KMPs) by TS divided by the total number of VMPs.

MS(P , TS , M) = # of KMPs / # of VMPs (3.6)

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 that is given in Algorithm 6 and 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.

The mutant policy generation algorithm is given in Figure 14. 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, 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.

As seen in Figure 14, a mutant policy is generated depending on the value of each cell. The function create_mutant copies all the rules of the original policy except the current rule, which is indicated by i, and changes the current rule depending on the corresponding cell value. Although create_mutant function can be simplified in terms of arguments to be passed and written as create_mutant(PT, i, false/true), is written with all cell values to be passed to indicate what happens to which cell in the create_mutant function.

```
Algorithm 6: Mutant Policy Generation Algorithm
input: PT
output: SMP
variables:
        PT: original firewall policy represented as a table
        SMP : set of mutant policies
steps:
        SMP := {}
        for i = 1 to number of rules
        for j = 1 to 6
        switch(j)
                Case 1 ; if (PT[i,1] = “tcp”) then
                        SMP := SMP ∪
                                create_mutant(PT, i, “udp”, PT[i,2], PT[i,3], PT[i,4], PT[i,5],false);
                                break;
                        else SMP := SMP ∪
                                create_mutant(PT, i, “tcp”, PT[i,2], PT[i,3], PT[i,4], PT[i,5],false);
                                break;
                Case 2 ; if (PT[i,2] of type concrete_IP_address) then
                        SMP := SMP ∪
                                create_mutant(PT, i, PT[i,1], PT[i,2], PT[i,3], PT[i,4], PT[i,5],false);
                                break;
                        else SMP := SMP ∪
                                create_mutant(PT, i, PT[i,1], PT[i,2], PT[i,3], PT[i,4], PT[i,5],true);
                                break;
                Case 3 ; if (PT[i,3] of type concrete_port ) then
                        SMP := SMP ∪
                                create_mutant(PT, i, PT[i,1], PT[i,2], PT[i,3], PT[i,4], PT[i,5],false);
                                break;
                        else SMP := SMP ∪
                                create_mutant(PT, i, PT[i,1], PT[i,2], PT[i,3], PT[i,4], PT[i,5],true);
                                break;
                Case 4 ; if (PT[i,4] of type concrete_IP_address ) then
                        SMP := SMP ∪
                                create_mutant(PT, i, PT[i,1], PT[i,2], PT[i,3], PT[i,4], PT[i,5],false);
                                break;
                        else SMP := SMP ∪
                                create_mutant(PT, i, PT[i,1], PT[i,2], PT[i,3], PT[i,4], PT[i,5],true);
                                break;
                Case 5 ; if (PT[i,5] of type concrete_port ) then
                        SMP := SMP ∪
```

Figure 14. Mutant Polciy Generation Algorithm (Cont. on next page)

```
                    create_mutant(PT, i, PT[i,1], PT[i,2], PT[i,3], PT[i,4], PT[i,5],false);
                    break;
                else SMP := SMP ∪
                    create_mutant(PT, i, PT[i,1], PT[i,2], PT[i,3], PT[i,4], PT[i,5],true);
                    break;
        Case 6 ;
                SMP := SMP ∪
                    create_mutant(PT, i, PT[i,1], PT[i,2], PT[i,3], PT[i,4], PT[i,5],true);
                    break;
end switch
end for
end for
return SMP
```

Figure 14. (cont.)

For j = 1, 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”.

For j = 2 or 4, 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.

For j = 3 or 5, 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.

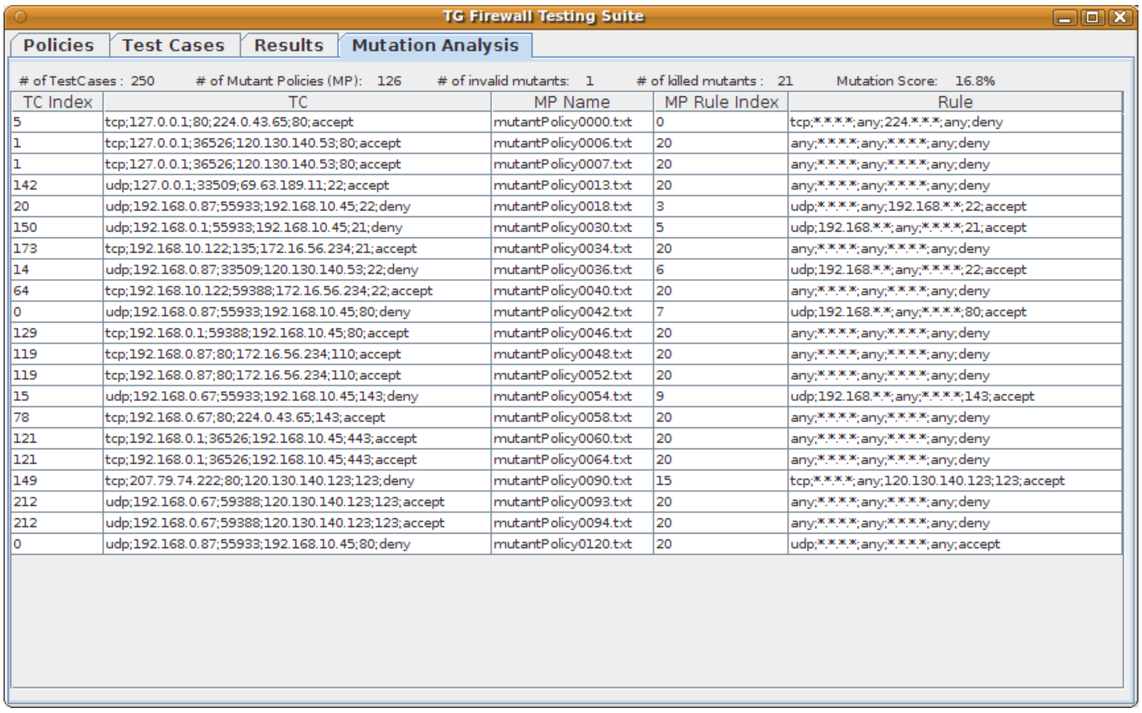
For j = 6, mutation operator is applied to the sixth cell. After mutant creation for each j index value, the resulting mutant policy is compared with the original policy. If they are the same policy, then the created mutant policy becomes an invalid mutant policy to be discarded, thus nil is returned. If they are different, then mutant policy returned from create_mutant function.

3.5. Implementation and Tool Support

For the implementation of our approach, 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, shown in Figure 16, 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 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.

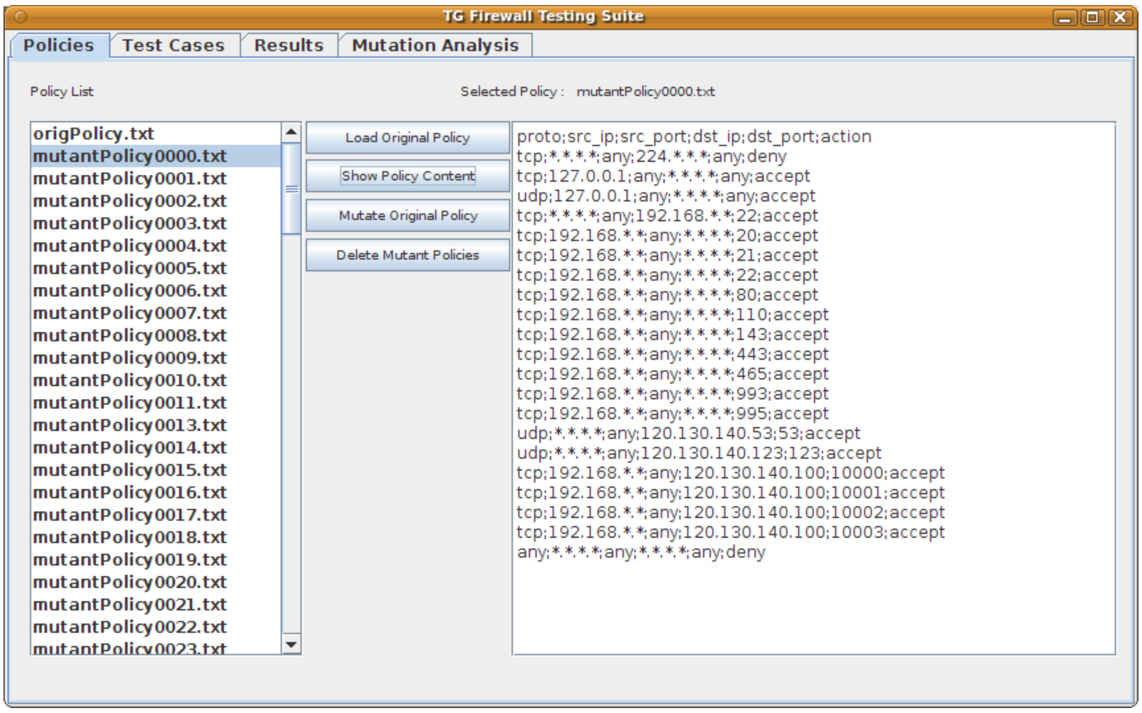
The last tabbed pane of TGFTS is called Mutation Analysis pane and shown in Figure 15. This pane is used to list killed mutant policies, the test case that killed the mutant and the rule of the mutant policies that the test case fails. As seen from the figure, number of test cases, number of mutant policies, number of invalid mutant policies, number of killed mutants, and calculated mutation score are presented at the top of Mutation Analysis tabbed pane.



# of TestCases : 250   # of Mutant Policies (MP): 126   # of invalid mutants: 1   # of killed mutants : 21   Mutation Score: 16.8%

TC Index	TC	MP Name	MP Rule Index	Rule
5	tcp;127.0.0.1;80;224.0.43.65;80;accept	mutantPolicy0000.txt	0	tcp;*.*.*.*;any;224.*.*.*;any;deny
1	tcp;127.0.0.1;36526;120.130.140.53;80;accept	mutantPolicy0006.txt	20	any;*.*.*.*;any;*.*.*.*;any;deny
1	tcp;127.0.0.1;36526;120.130.140.53;80;accept	mutantPolicy0007.txt	20	any;*.*.*.*;any;*.*.*.*;any;deny
142	udp;127.0.0.1;33509;69.63.189.11;22;accept	mutantPolicy0013.txt	20	any;*.*.*.*;any;*.*.*.*;any;deny
20	udp;192.168.0.87;55933;192.168.10.45;22;deny	mutantPolicy0018.txt	3	udp;*.*.*.*;any;192.168.*.*;22;accept
150	udp;192.168.0.1;55933;192.168.10.45;21;deny	mutantPolicy0030.txt	5	udp;192.168.*.*;any;*.*.*.*;21;accept
173	tcp;192.168.10.122;135;172.16.56.234;21;accept	mutantPolicy0034.txt	20	any;*.*.*.*;any;*.*.*.*;any;deny
14	udp;192.168.0.87;33509;120.130.140.53;22;deny	mutantPolicy0036.txt	6	udp;192.168.*.*;any;*.*.*.*;22;accept
64	tcp;192.168.10.122;59388;172.16.56.234;22;accept	mutantPolicy0040.txt	20	any;*.*.*.*;any;*.*.*.*;any;deny
0	udp;192.168.0.87;55933;192.168.10.45;80;deny	mutantPolicy0042.txt	7	udp;192.168.*.*;any;*.*.*.*;80;accept
129	tcp;192.168.0.1;59388;192.168.10.45;80;accept	mutantPolicy0046.txt	20	any;*.*.*.*;any;*.*.*.*;any;deny
119	tcp;192.168.0.87;80;172.16.56.234;110;accept	mutantPolicy0048.txt	20	any;*.*.*.*;any;*.*.*.*;any;deny
119	tcp;192.168.0.87;80;172.16.56.234;110;accept	mutantPolicy0052.txt	20	any;*.*.*.*;any;*.*.*.*;any;deny
15	udp;192.168.0.67;55933;192.168.10.45;143;deny	mutantPolicy0054.txt	9	udp;192.168.*.*;any;*.*.*.*;143;accept
78	tcp;192.168.0.67;80;224.0.43.65;143;accept	mutantPolicy0058.txt	20	any;*.*.*.*;any;*.*.*.*;any;deny
121	tcp;192.168.0.1;36526;192.168.10.45;443;accept	mutantPolicy0060.txt	20	any;*.*.*.*;any;*.*.*.*;any;deny
121	tcp;192.168.0.1;36526;192.168.10.45;443;accept	mutantPolicy0064.txt	20	any;*.*.*.*;any;*.*.*.*;any;deny
149	tcp;207.79.74.222;80;120.130.140.123;123;deny	mutantPolicy0090.txt	15	tcp;*.*.*.*;any;120.130.140.123;123;accept
212	udp;192.168.0.67;59388;120.130.140.123;123;accept	mutantPolicy0093.txt	20	any;*.*.*.*;any;*.*.*.*;any;deny
212	udp;192.168.0.67;59388;120.130.140.123;123;accept	mutantPolicy0094.txt	20	any;*.*.*.*;any;*.*.*.*;any;deny
0	udp;192.168.0.87;55933;192.168.10.45;80;deny	mutantPolicy0120.txt	20	udp;*.*.*.*;any;*.*.*.*;any;accept

Figure 15. TG Firewall Testing Suite, Mutation Analysis Pane



Selected Policy : mutantPolicy0000.txt

```
origPolicy.txt
mutantPolicy0000.txt
mutantPolicy0001.txt
mutantPolicy0002.txt
mutantPolicy0003.txt
mutantPolicy0004.txt
mutantPolicy0005.txt
mutantPolicy0006.txt
mutantPolicy0007.txt
mutantPolicy0008.txt
mutantPolicy0009.txt
mutantPolicy0010.txt
mutantPolicy0011.txt
mutantPolicy0013.txt
mutantPolicy0014.txt
mutantPolicy0015.txt
mutantPolicy0016.txt
mutantPolicy0017.txt
mutantPolicy0018.txt
mutantPolicy0019.txt
mutantPolicy0020.txt
mutantPolicy0021.txt
mutantPolicy0022.txt
mutantPolicy0023.txt
```

```
proto;src_ip;src_port;dst_ip;dst_port;action
tcp;*.*.*.*;any;224.*.*.*;any;deny
tcp;127.0.0.1;any;*.*.*.*;any;accept
udp;127.0.0.1;any;*.*.*.*;any;accept
tcp;*.*.*.*;any;192.168.*.*;22;accept
tcp;192.168.*.*;any;*.*.*.*;20;accept
tcp;192.168.*.*;any;*.*.*.*;21;accept
tcp;192.168.*.*;any;*.*.*.*;22;accept
tcp;192.168.*.*;any;*.*.*.*;80;accept
tcp;192.168.*.*;any;*.*.*.*;110;accept
tcp;192.168.*.*;any;*.*.*.*;143;accept
tcp;192.168.*.*;any;*.*.*.*;443;accept
tcp;192.168.*.*;any;*.*.*.*;465;accept
tcp;192.168.*.*;any;*.*.*.*;993;accept
tcp;192.168.*.*;any;*.*.*.*;995;accept
udp;*.*.*.*;any;120.130.140.53;53;accept
udp;*.*.*.*;any;120.130.140.123;123;accept
tcp;192.168.*.*;any;120.130.140.100;10000;accept
tcp;192.168.*.*;any;120.130.140.100;10001;accept
tcp;192.168.*.*;any;120.130.140.100;10002;accept
tcp;192.168.*.*;any;120.130.140.100;10003;accept
any;*.*.*.*;any;*.*.*.*;any;deny
```

Figure 16. TG Firewall Testing Suite, Policies Pane

CHAPTER 4

CASE STUDY

The policy of the firewall is considered as a specification. Therefore, the firewall testing context in this work 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. Even if there is no mismatch between specified and loaded policies, 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 work, which is another merit of the proposition.

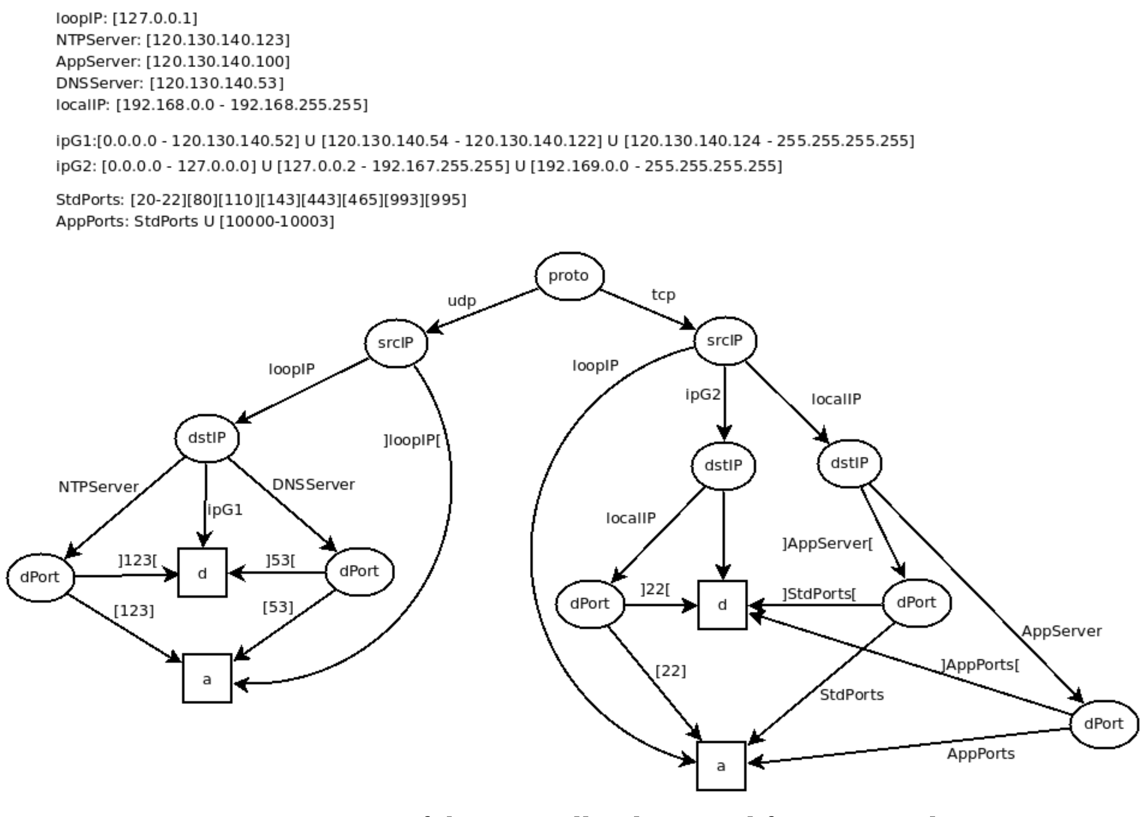
4.1. Firewall Policy Under Consideration

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. A mutant of it can be seen in Figure 16. Then the policy is converted to a FDD, which is presented in Figure 17. 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 Figure 17, the abstract test case is as follows:

Table 3. Abstract Test Case

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 1 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]
proto
udp
tcp
srcIP
loopIP
]loopIP[
dstIP
NTPServer
ipG1
DNSServer
dPort
]123[
d
]53[
[123]
[53]
a
srcIP
loopIP
ipG2
localIP
dstIP
localIP
]AppServer[
]22[
]StdPorts[
[22]
]AppPorts[
StdPorts
AppServer
AppPorts

Figure 17. 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 and injected to the network.

4.2. Results and Discussion

The PUC for the case study has 21 rules. We generate 126 mutant policies from PUC using mutant policy generation method. 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 WTSi, 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 Figure 18.

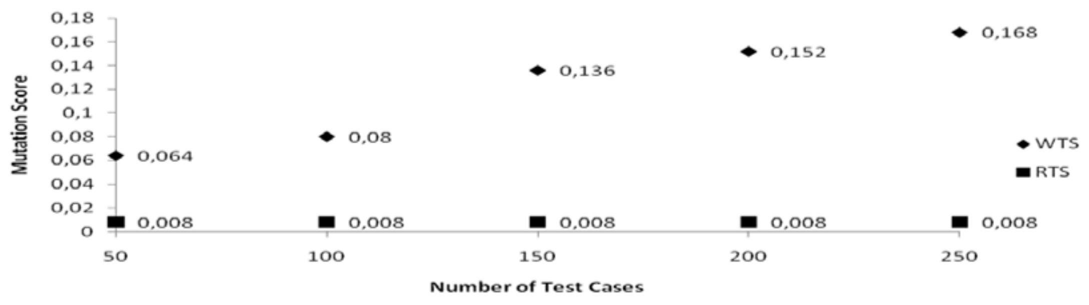


Figure 18. Mutation Scores of WTS and RTS Sets

Moreover, another five test sets, called RTSi, 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 Figure 18 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 (Li & Malaiya, 1994). However, that certain point depends on the operational profile of the software and may be computationally infeasible.

CHAPTER 5

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.

BIBLIOGRAPHY

Ammann, P., & Black, P. (1999). A Specification-Based Coverage Metric to Evaluate Test Sets. *14th IEEE International Symposium on High-Assurance Systems Engineering*. Washington, D.C.

Committee, C. -S. (2013, 06). *Software and Systems Engineering Standards.* Retrieved from IEEE Standarts: http://standards.ieee.org/findstds/standard/software_and_systems_engineering.html

El-Atawy, A., Ibrahim, K., Hamed, H., & Al-Shaer, E. (2005). Policy segmentation for intelligent firewall testing. *1st IEEE ICNP Workshop on Secure Network Protocols (NPSec).*

Gouda, M., & Liu, X. (2004). Firewall Design: Consistency, Completeness, and Compactness. *24th International Conference on Distributed Computing Systems.*

Gupta, A., & Jalote, P. (2008). An Approach for Experimentally Evaluating Effectiveness and Efficiency of Coverage Criteria for Software Testing. *International Journal on Software Tools for Technology Transfer*, 145-160.

Hwang, J. H., Xie, T., Chen, F., & Liu, A. X. (2008). Systematic Structural Testing of Firewall Policies. *IEEE Symposium on Reliable Distributed Systems.* Naples.

Jia, Y., & Harman, M. (2010). An Analysis and Survey of the Development of Mutation Testing. *IEEE Transactions on Software Engineering*.

Kamara, S., Fahmy, S., Schultz, E., Kerschbaum, F., & Frantzen, M. (2003). Analysis of Vulnerabilities in Internet Firewalls. *Computers&Security*, 214-232.

Kaner, C. (2006). Exploratory Testing. *Quality Assurance Institute Worldwide Annual Software Testing Conference.* Orlando, FL.

Li, N., & Malaiya, Y. (1994). On Input Profile Selection for Software Testing. *5th International Symposium on Software Reliability Engineering.* CA.

Ma, H. (2004). Specification Based Firewall Testing. *Master of Arts Thesis*. San Marcos: Texas State University.

Mathur, A. P. (2008). *Foundations of Software Testing: Fundamental Algorithms and Techniques.* Pearson Education.

Moses, T. (2005). *Extensible access control markup language (xacml) version 2.0.* Oasis Standard.

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

Oppliger, R. (1997, May). Internet Security: FIREWALLS and BEYOND. *Communications of the ACM Vol. 40, No. 5*, pp. 92-102.

Sabnani, K., & Dahbura, A. (1988). A Protocol Test Generation Procedure. *Computer Networks and ISDN Systems*, 285-297.

Senn, D., Basin, D., & Caronni, G. (2005). Firewall Conformance Testing. *Testing of Communicating Systems* (pp. 226-241). Springer.

Smith, B., & Williams, L. (2009). Should Software Testers Use Mutation Analysis to Augment a Test Set? *Journal of Systems and Software*, 1819-1832.

Tuglular, T. (2007). Test Case Generation for Firewall Implementation Testing Using Software Testing Techniques. *International Conference on Security of Information and Networks.* Cyprus.

Tuglular, T., & Belli, F. (2008). Model Based Mutation Testing of Firewalls. *Testing: Academic and Industrial Conference-Practice and Research Techniques.* UK.

Tuglular, T., & Belli, F. (2009). Directed Acyclic Graph Modeling of Security Policies for Firewall Testing. *1st International Workshop on Model-Based Verification & Validation.* Shanghai.

Tuglular, T., & Belli, F. (2009). Protocol-Based Testing of Firewalls. *4th South-East European Workshop on Formal Methods.* Thessaloniki.

Tuglular, T., & Gercek, G. (2010). Feedback Control Based Test Case Instantiation for Firewall Testing. *7th International Workshop on Software Cybernetics.* Seoul.

Wimmel, G., & Jürjens, J. (2002). Specification-Based Test Generation for Security-Critical Systems Using Mutations. *Formal Methods and Software Engineering*, (pp. 471-482).

Zaugg, G. (2005). Firewall Testing. *MA Thesis*. Zurich: Swiss Federal Institute of Technology.