An Automated Framework for Validating Firewall Policy Enforcement

Adel El-Atawy[*], Taghrid Samak[*], Zein Wali[*], Ehab Al-Shaer[*],Sheng Li[†]
[*]School of Computer Science, Telecommunication, and Information Systems
DePaul University
Chicago, Illinois 60604
Email: {aelatawy, taghrid, zwali, ehab}@cs.depaul.edu
[†]Cisco
San Jose, California 95134
Email: {fclin, chpham, sheli}@cisco.com

Abstract

The implementation of network security devices such as firewalls and IDSs are constantly being improved to accommodate higher security and performance standards. Using reliable and yet practical techniques for testing the functionality of firewall devices particularly after new filtering implementation or optimization becomes necessary to assure proven security. Generating random traffic to test the functionality of firewall matching is inefficient and inaccurate as it requires an exponential number of test cases for a reasonable coverage. In addition, in most cases the policies used during testing are limited and manually generated representing fixed policy profiles.

In this paper, we present a framework for automatic testing of the firewall policy enforcement or implementation using efficient random traffic and policy generation techniques. Our framework is a two-stage architecture that provides a satisfying coverage of the firewall operational states. A large variety of policies are randomly generated according to custom profiles and also based on the grammar of the access control list. Testing packets are then generated intelligently and proportional to the critical regions of the generated policies to validate the firewall enforcement for such policies. We describe our implementation of the framework based on Cisco IOS, which includes the policy generation, test cases generation, capturing and analyzing firewall output, and creating detailed test reports. Our evaluation results show that the automated security testing is not only achievable but it also offers a dramatically higher degree of confidence than random or manual testing.

I. Introduction

Firewalls work as filtering devices at the boundary of different (sub)networks based on policies or access control lists (ACL). They are important security devices as they protect the internal network from attacks and unauthorized traffic. Due to the persistent effort to enhance and optimize firewall protection and performance respectively, firewalls undergo continuous modification to their internal design and implementation to deploy new optimized filtering algorithms, or adding new features into the ACL as firewall policies evolves syntactically and semantically. This increases the chance of software bugs that might invalidate the filtering decision, thereby causing security violations. Therefore, firewalls devices with new software releases require thorough validation to uncover errors in the implementation or malfunctioning components.

The problem of testing a firewall has two stages:

- Generating random policies (*i.e.*, ACLs) with different configurations such as rule complexity, rule interaction, filtering criteria, etc, and
- generating packets to test the implementation of the device under test (DUT) using these policies.

Both problems must be addressed to claim that every aspect of the firewall enforcement is validated. Our framework handles both problems using intelligent policy generation and policy segmentation techniques.

The problem of policy generation has two sides of its own. The first one, is to generate rules that conform with the syntax specification of the firewall. The other side of the problem is to generate rules that use different field values, different complexities (*i.e.*, number of optional parameters/fields used) and covering a wide range of rule inter-relations (*i.e.*, overlapping, super/subsets, etc). The former problem is solved by specifying the syntax using a tailored form of augmented context-free grammar. The latter is addressed by generating the rules based on a neutral representation (*i.e.*, a finite state automaton accepting the grammar), thus separating the policy generation logic from the specifics of the language. Using this approach, both aspects of the problem are targeted simultaneously and policies with a customizable nature can be generated following any user-specified ACL syntax.

Testing the firewall by exhaustively injecting all possible packets into the firewall will be enough. It is a simple operation to enumerate all possible values in the packet header fields, but it is not feasible due to the number of packets needed (even if we restricted these packets to ranges with relevant addresses and ports). Restricting the domain even further (by confining packets to realistic values and tuning the encoding) will only reduce the required testing time from about 4×10^{13} years for the complete address space, to 4.5 years (using one billion packets per second). As seen in the Table I, the savings can be huge but the needed time is still prohibitive. Random sampling can be used but its one-sided error probability (*i.e.*, probability of faulty firewall passes the test, or having a false-negative) is impractically high. Therefore, we introduce the concept of policy segmentation in order to achieve a smart criteria for packet selection.

Span method	Number of values	Time on a 1G packet/sec system. (years)
Entire traffic space	$1.2676x10^{30}$	$4.0169x10^{13}$
Only relevant traffic (using class C network)	$7.555x10^{22}$	$2.394x10^{6}$
Using some optimizations	$1.44x10^{17}$	4.566

TABLE I

REQUIRED TIME TO EXHAUSTIVELY TEST DIFFERENT ADDRESS SPACES

In the next section, an overview of related work is provided. Section III discusses the system framework and its different modules. The policy generation is discussed in section IV. The test data generator will be discussed in section V. The reporting capabilities of the framework will be shown in section VI. In section VII the system will be evaluated, followed by conclusion and future work.

II. Related Work

Product testing is considered not only the most important but also the most expensive and time consuming phase in any production line. Testing techniques are categorized into: black-box “functional” and white-box “structural” testing. In black-box testing, the knowledge of the internal structure/implementation of the component under test is not used or in most cases it is not even known to the tester or hidden by the component owner. This test can be done exhaustively but in most cases the samples needed to perform the test is impractically large. Therefore, only a subset of the input data is selected. The data selection is done from the input domain boundaries and within each domain the sample data is selected statistically [2], [5]. In white-box testing, the internal structure or the implementation must be known because the whole test is based on this knowledge where the samples are selected to test all possible cases or states in the implementation [2], [4].

Mostly, network devices (like switches, routers, network balancers and firewalls) are tested using the black-box ”functional” testing concept. Usually, this test is divided into sub-tests: auditing, build verification, availability, manageability and security tests [8].

The firewall, as a network device, follows the previous testing procedure but due to its nature it has a little different testing approach. Firewall errors can be categorized into four error types: security, implementation, policy and capability errors [11]. Each error type has its own detection techniques. Generally, we can categorize the firewall testing techniques into two approaches, theoretical approaches [7], [10], [12] and practical approaches [11], [3], [6].

In the theoretical approach, the testing is based on a formal model of the firewall and the surrounding network. Vigna proposes in [10] a firewall testing methodology based on a formal model of networks that allows the test engineer to model the network environment of the firewall. This allows to formally prove that the topology of the network provides the needed protection against attacks. Also, Jurjens et al in [7] used a CASE tool to model

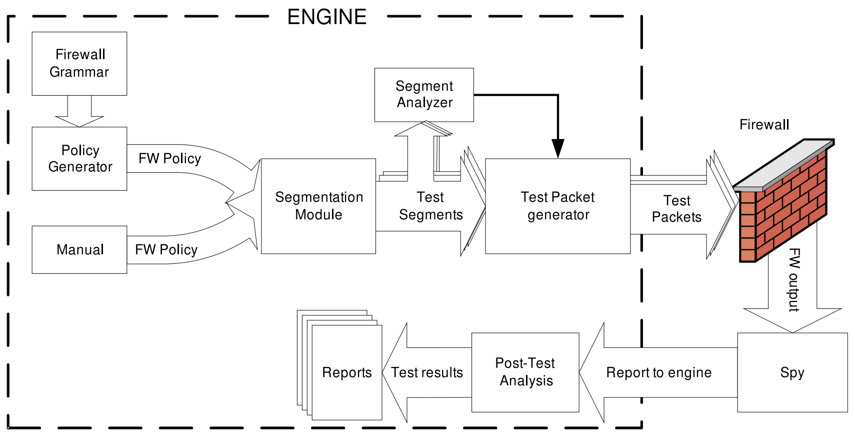


Fig. 1. System framework: A high level view

the firewall and the surrounding network then test cases are generated to test the firewall against a list of security vulnerabilities. Others, in the same camp, like Wool [12] introduced a tool capable of analyzing configuration parameters from real environments. This firewall analyzer was based on the work of Mayer, Wool and Ziskind [1]. The analyzer is fed with the firewall configuration files and the firewall routing table then it uses this information to simulate the firewall behavior. The simulation is done totally off-line without sending any packets.

In the practical approach: most of the previous work provide methodologies to perform penetration testing against the firewall. Ranum [9] identifies two kinds of testing methods: checklist and design-oriented testing. Checklist testing is equivalent to penetration testing in running a list of vulnerability scanners against the firewall. Design-oriented testing is quite different; we ask who implemented the firewall “why do they think the firewall will protect the network (or not)” and based on their answers a set of test cases are designed to check their claims. In [3] the authors present a methodology to test the firewall security vulnerability using two tests one is automated using a security analysis tool called SATAN while the other test is manual which is based on interacting with the firewall. Haeni in [6] describes a methodology to perform firewall penetration testing. The test was structured in four steps, indirect information collection, direct information collection, attack from the outside and attack from the inside. In ICSA labs, a firewall is being tested against a pre-defined set of firewall violations (*i.e.*, firewall logging capability, firewall security vulnerability, firewall policy, etc.) which are described by Walsh in [11] and the one that passes their test get certified.

Most of the previously mentioned efforts tried to address this problem from different points of view but no work was published about testing whether the firewall implements the policy correctly or not. Also, none of the above mentioned tools and models generate real policies that can set different environments for the firewall to operate under.

III. System Framework

An external view of the system shows three components: The Engine, the Spy, and the User Interface. The Engine is the core and it is where most of the processing takes place. The Spy resides behind the firewall to report back to the Engine how the firewall handled the traffic, and the User Interface is a light weight front end to the engine.

From a design point of view, the architecture of our system consists of the following main modules: Policy generator, Segmentation/analysis module, Test packet generator, post-test analysis module. Other helper modules include the BNF parser, traffic injection module, policy compiler, and the spy process (see Fig 1).

A typical test cycle starts with specifying all parameters for policy (*e.g.*, grammar, policy size, etc) and traffic generation (*e.g.*, number of packet, injection rate, etc), and system configuration (*e.g.*, firewall type and address). Then, when the test is initiated, the testing engine will generate a policy, analyze the policy for traffic selection and load it into the firewall. The digested policy (in the form of segments, will be discussed in detail in section V-B) will be used to generate packets that will be in turn injected to the firewall. The outcome of the firewall will be

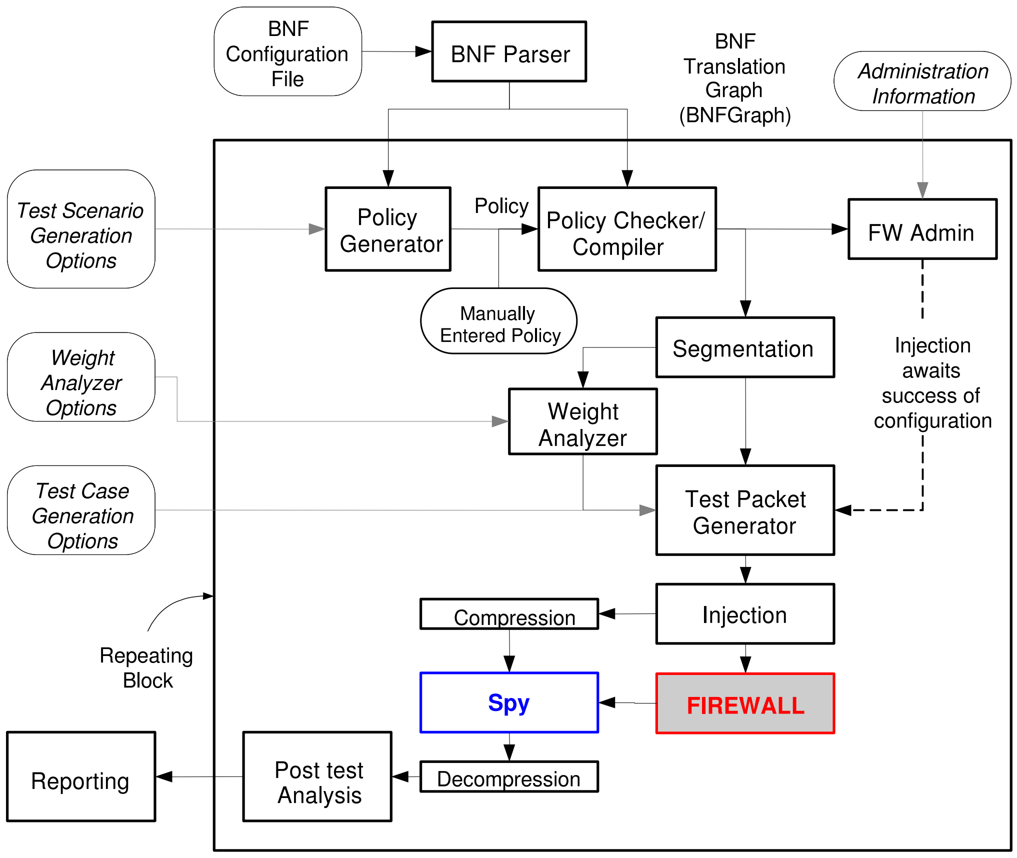


Fig. 2. Detailed System Architecture

monitored by the Spy process. Afterwards, the resulting packets will be analyzed and a report will be generated. The cycle of policy generation and traffic injection can be repeated as the administrator wishes. A more detailed diagram showing the exact implementation of the system is shown in Fig 2. The following is a description of each component:

- **BNF Parser (Policy Grammar Parser):** This core module reads the grammar specification of the Device Under Test (DUT), and builds the Finite State Automaton (FSA) that accepts this grammar. The grammar is provided in an augmented BNF format. All relations and restrictions on field values and among fields are incorporated into this specification.
- **Policy Generation Module** This module is responsible for generating different policies according to a set of parameters (including the grammar specification) as provided by the user. These policies would be used to configure the firewall for the test. Optionally, the user can load policies manually overriding this module functionality. The output of this module is a policy in plain text that follows the firewall specific syntax.
- **Policy Parsing, Checking and Compiling** This module parses a policy, and compiles it into an internal representation of constraints on packet header field bits.
- **Segmentation Module** The segmentation algorithm builds a list of segments that is equivalent to the compiled policy. Using the rule-rule intersections and overlaps, different areas are identified, and the segments are formed. See section V-B for details.
- **Segment Weight Analyzer** The segments are analyzed and each is assigned a weight based on a number of factors such as the number of rules intersecting in the segment and the size of the segment address space. The segment weight is a measure of the criticality of the segment.
- **Test Packet Generator** This module generates test packets distributed in proportion to the segment weight. For example, segments that involve many interacting rules will potentially receive more test packets than others.

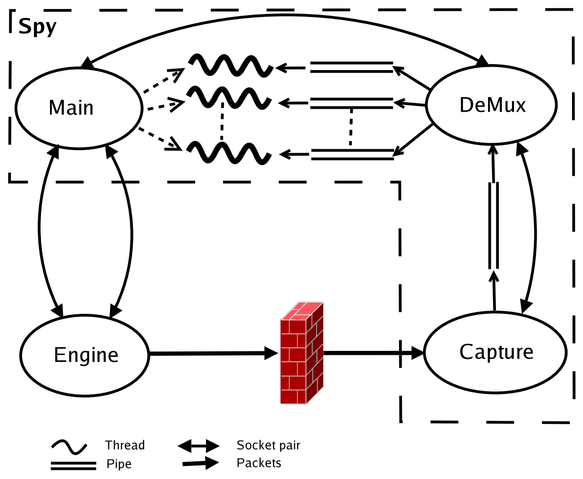


Fig. 3. The internal design of the Spy module. Its interaction with the firewall and Engine is illustrated

Each test packet carries test case information that includes a packet sequence number, the test session ID, and a signature to avoid confusion with other cross-traffic packets.

- **Spy: firewall monitoring tool** The Spy is a stand alone program that resides behind the firewall. It sniffs all outgoing packets, and collects them based on the Engine's request. When a test starts, the spy is informed by the test session ID. All packets are tested for this value, and if found they are marked in a bitmap. When the test is over, the Spy will compare the expected with the actual firewall behavior and errors if any will be sent back to the engine for analysis. The bitmap that represents the expected behavior (sent from the Engine to the Spy prior to the test), and the bitmap of the discrepancies (sent from the Spy to the Engine after the test) are both compressed to save communication overhead.
  The Spy is built to manage several tests simultaneously, each can be coming from a different Engine. Thus, it has to be designed to support such a high traffic volume, without dropping packets due to overload (design shown in Fig 3). It is split into three processes; (1) Capture process: reads all packets from the media, and forwards packets that has the framework signature into the next process, (2) DeMux process: takes all test packets that were captured, and demultiplexes them into their corresponding threads, and (3) The Main Process: that communicates with the Engine, receives the test requests, creates a thread for each test, creates the appropriate pipes/queues for these threads, and informs the DeMux process about the new test ID's.
- **Post-Test Analyzer** The collected information (by the spy) are then returned to the Engine where analysis is performed and a report is generated that shows the errors (if any). Several hints are extracted showing where and why the errors might have occurred. This is provided to the user as a distribution of errors and their types over different segments, rules, and fields.

IV. Policy Generation

In this section, we present the design details of the modules responsible for the first phase of the testing cycle; namely, Policy Generation. The modules involved are the BNF grammar parser, the policy checker/compiler, and the policy generation module.

A. Policy Grammar

Describing a general grammar that can accommodate all possible access-control list syntaxes is not as simple as it seems. Specifying only the syntax can be a headache but a straightforward task. However, specifying the semantics of the different clauses in the grammar (*i.e.*, how each of these tokens and clauses interact to form the final ACE constraint that correspond to its actual effect in filtering traffic) is another problem. Extra embedded annotations are needed to describe these semantics to the grammar parser and the policy generator afterwards.

Among the annotations needed: which field is this clause/token configuring? If this is a literal, does it have a special meaning (*e.g.*, "accept", "permit", "any")? Can this value be represented by a literal, can we use lookup

```
S := "access-list" Policy-Number action SrcAddr [opt1]
Policy-Number\FieldID(100) := \number(1,99)
action\FieldID(0) := "permit"\V(1) | "deny"\V(0)
SrcAddr\FieldID(2) := IPany | IPpair
IPany  := "any"\Trans("IP","op","any")
IPpair := \IPvalue\Trans("IP","op","IPsubnet") [OptMask]
OptMask := \IPmask\Trans("IP","op","IPmask")
opt1\FieldID(80) := "log"\V(1)
```

Fig. 4. A simple access-control list syntax

files for this task (*e.g.*, protocol and port names)? Is this token applicable for all protocols (*e.g.*, ICMP qualifiers are only valid with ICMP specified as the protocol, the same goes for TCP extra control bits)?

From a theoretical point of view, adding all of the annotations needed for the above mentioned tweaks will not result in a more powerful grammar. It will still be a CFG (context free grammar). This observation is necessary in order to guarantee that a simple PDA (PushDown Automaton) can recognize this grammar successfully (*i.e.*, accepts any policy that follows this grammar). Simply, this is dealt with a graph (represents the finite state automaton) and the ordinary stack of recursive calls for the accompanying stack required to convert the FSA to a PDA.

An example of a simple access-list syntax that supports constraints only for the source address can be written as follows:

The grammar has two sets of symbols; terminal and non-terminal symbols. Some special terms are included in the grammar to express the semantics and field relations. In our implementation, some of the general properties of the grammar are:

- Literal terminal symbols are enclosed with double quotes and are an exact string that will appear in the rule definition.
- Non-terminal symbols are defined by following lines in the grammar.
- Each line defines a non-terminal.
- Square brackets ”$[]$” enclose optional fields.
- Each symbol followed by a “$\backslash$“ represents an operation on the field that will be performed by the parser.
- Round brackets ”$()$” are used to hold parameters for operations.
- Symbols starting by a “$\backslash$“ are special fields that are predefined in the parser.

Special keywords/operations in the grammar are:

- $\backslash$*FieldID(n)* Indicates the field number (n) to be used in inter-field dependency reference. Those numbers are predefined and will be used to retrieve the physical position of the field in later stages.
- $\backslash$*num(n_1,n_2)* An integer range from n_1 to n_2.
- $\backslash$*V(x)* The value of the current token is x. For example, “tcp”$\backslash$V(6) means the word “tcp” has the value of 6.
- *IPvalue* Special handling for IPs, in parsing and generation.
- *IPmask* Special handling for subnet masks, in parsing and generation.
- *Trans* This is required to specify non-simple handling of data in the rule. For example, conditions with inequalities over port values, and the subnet mask effect of IP addresses.

The parser deals with the symbols associated with the *Trans* operator according to its following parameters. For each such symbol, the context must be defined. The way the parser should deal with the symbol should be given as well. The parser has a set of predefined methods that will be used to perform the operation over the field. Three parameters must be provided to the *Trans* operator. The first parameter refers to the context of the symbol (IP, port, protocol). The second parameter corresponds to the type of the last parameter which is the action to apply.

In the same sense a more complex grammar is defined in Fig 5. To be able to accommodate the “extended access list” features, some extra keywords have been added. For example, $\backslash Lookup$, and $\backslash Cond$ are needed to specify extra options. The former is used in cases where a list of constants has no special handling but their value, so they are retrieved from a file. Typical use can be for port and protocol names (instead of listing them using the “$\backslash$V” keyword). The latter is used for these clauses that only appear when another field has been given a certain value. This takes places more often when clauses depend on the protocol value. It is associated with a non-terminal, and

```
S   := "access-list" Policy-Number action protocol SrcAddr RT [Logging]
RT :=  RT1 | RT2
RT1 := [SrcPort] DestAddr L3Extra RL4
RT2 := DestAddr L3Extra RT2b
RT2b := RL4 | Fragments
L3Extra   := [Prec] [Tos]
RL4    :=  [DestPort] [FIN] [PSH] [SYN] [URG] [AckRstEst] [igmpquals] [icmpquals]
AckRstEst :=  [ACK] [RST] | [Established]

Policy-Number\FieldID(100)  := \number(100,199)
action\FieldID(0)   := "permit"\V(1) | "deny"\V(0)
protocol\FieldID(1)     := \number(0,255) | \Lookup("number","protocols.txt")
SrcAddr\FieldID(2)  := IPaddr
DestAddr\FieldID(3)     := IPaddr
SrcPort\FieldID(4)  := Port
DestPort\FieldID(5)     := Port

IPaddr  := IPany | IPhost | IPpair
IPany       := "any"\Translate("IP","operator","any")
IPhost      := "host" \IPvalue\Translate("IP","operator","IPhost")
IPpair      := \IPvalue\Translate("IP","operator","IPsubnet") \IPmask\Translate("IP","operator","IPmask")

Port \Cond(1,17)    := PortOp1|PortOp2|PortOp3|PortOp4|PortOp5
Port \Cond(1,6)     := PortOp1|PortOp2|PortOp3|PortOp4|PortOp5
PortOp1         := "eq" Y\Translate("port","operator","eq")
PortOp2         := "lt" Y\Translate("port","operator","lt")
PortOp3         := "range" Y\Translate("port","operator","ge") Y\Translate("port","operator","le")
PortOp4         := "gt" Y\Translate("port","operator","gt")
PortOp5         := "neq" Y\Translate("port","operator","ne")
Y \Cond(1,17)   := \number(0,65535) | \Lookup("number","udpports.txt")
Y \Cond(1,6)    := \number(0,65535) | \Lookup("number","tcpports.txt")

icmpquals\FieldID(6)\Cond(1,1)  := \number(0,255) \number(0,255) | \Lookup("number","icmpquals.txt")

igmpquals\FieldID(19)\Cond(1,2) := \number(0,15) | Lookup("number","igmpquals.txt")

Prec\FieldID(7)         := "precedence" \number(0,7)
Tos\FieldID(8)          := "tos" \number(0,15)

Logging\FieldID(80)             := "log"\V(1)

Established\Cond(1,6)\FieldID(9)    := "established"
ACK\FieldID(11)\Cond(1,6)       := "ack\V(1)
FIN\FieldID(12)\Cond(1,6)       := "fin"\V(1)
PSH\FieldID(13)\Cond(1,6)       := "psh"\V(1)
RST\FieldID(14)\Cond(1,6)       := "rst"\V(1)
SYN\FieldID(15)\Cond(1,6)       := "syn"\V(1)
URG\FieldID(16)\Cond(1,6)       := "urg"\V(1)

Fragments\FieldID(10)           := "fragments"\V(1)
\\EndRules
```

Fig. 5.   A more complex access-control list syntax

given in the form of $Cond(ID, v)$. This translates to: this grammar line is only defined if field ID was assigned the value v.

B. BNF Graph

The BNF graph is a directed graph corresponding to the FSA of the grammar. The graph has a starting state, which has no incoming edges, and a final state, which has no outgoing edges. The final state is the only accepting state in the graph. In between those nodes, the graph is generated according to the given grammar syntax. The graph is used for parsing, checking and compiling input policies to the system. It is also used in the process of policy generation. In this section the structure of the graph is presented. The following section explain how this graph is used for policy generation.

Each node represents a field in the grammar (or a symbol). Outgoing edges from a graph node control the transition between nodes, according to values allowed for the corresponding field. The links also store information

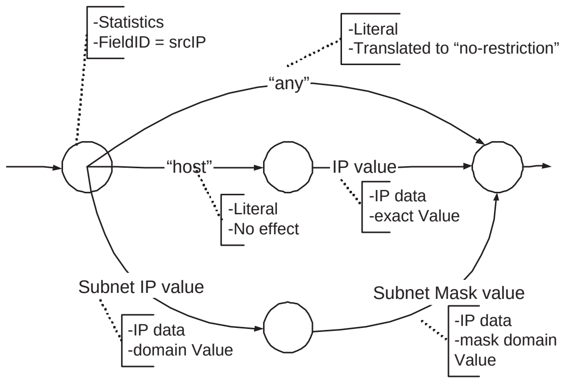


Fig. 6. A subsection of the BNF graph showing the IP section. Node type is described beside each one.

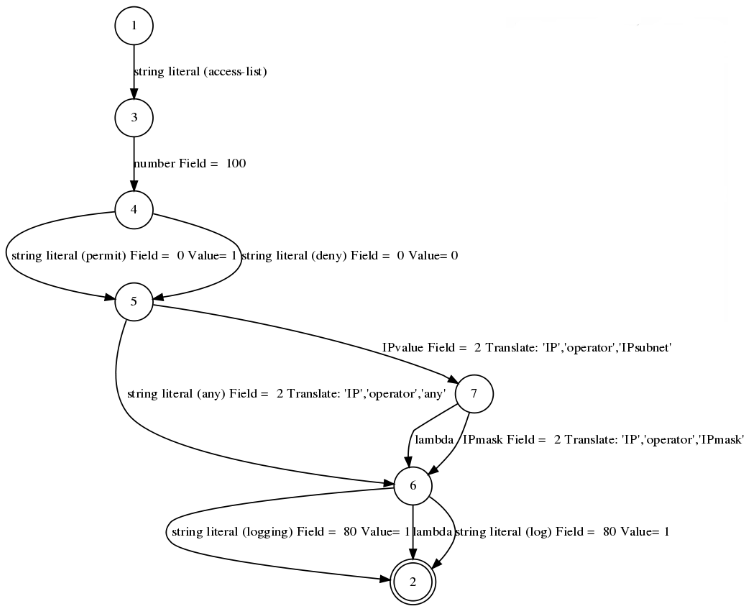

1
string literal (access-list)
3
number Field = 100
4
string literal (permit) Field = 0 Value= 1
string literal (deny) Field = 0 Value= 0
5
IPvalue Field = 2 Translate: 'IP','operator','IPsubnet'
string literal (any) Field = 2 Translate: 'IP','operator','any'
7
lambda
IPmask Field = 2 Translate: 'IP','operator','IPmask'
6
string literal (logging) Field = 80 Value= 1
lambda
string literal (log) Field = 80 Value= 1
2

Fig. 7. A graph representing the basic grammar example.

about field IDs, dependency conditions between fields, and other annotations provided in the grammar.

Rules of a given policy are parsed using this graph. For each rule, the symbols are separated by blank spaces. The process begins at the start state, then each encountered symbol is matched against outgoing edges/links from the current node. Reaching the final state successfully results in accepting the rule.

Following is a summary of the data stored at each edge/link:

- *String* The string to be matched.
- *Value* Value (or range of values) of the given string.
- *Field* The field affected by the current link.
- *Condition* Dependencies between fields. This includes the controlling field ID, and the value that this field should have for the link to be active.
- *Probability* A transition probability. This part is used only for policy generation.

Figure 6 shows the portion of the graph corresponding to the IP. Also, Fig. 7 shows the whole graph for the simple grammar example.

C. *The Generation Process*

A complete policy consists of an ordered set of rules. The syntax of each rule is defined by a given BNF grammar. The corresponding graph is generated from the grammar. A complete traversal of the graph from the start state to the final accepting state, is equivalent to a single rule. The graph is traversed according to the probabilities specified at each link. For each generated policy, the following parameters control the configuration of the policy:

- *Policy size* The average number of rules that will be generated for each policy.
- *Average rule complexity* This parameter holds a percentage $0\% - 100\%$ that maps to the probability to use an optional field. The higher the value the more complex the rule will be.
- *Probability of accept* The probability of choosing the action of the rule to be permit.

Another set of parameters are used to control the ranges of the field values. For example, the generation should be able to favor the lower range of port values. Also, exhausting a set of values for a specific field is possible.

V. Smart Traffic Generation

In this section we will consider the method used for generating testing traffic for the firewall. The main goal is to generate the least amount of traffic that is needed to test the possible decision paths for the given firewall policy. In order to have this span over the firewall/policy behavioral cases, we will have to define the whole space over which the policy could be tested.

Definition 1:

Traffic Address Space The space whose elements are all the different tuples that identify traffic in a network environment, especially from the firewall point of view. Normally this is composed of <protocol, source address, source port, destination address, destination port>.

Our test has to be guided into the traffic space in a way to guarantee intelligent selection of samples.

A. *Firewall Rules Representation*

Rules and traffic address subspaces are represented using Boolean expressions. All Packets belonging to a certain rule must satisfy its boolean expression. Such representation will make the operations mentioned in following sections more clear and efficient. The expression is created by assigning a variable to each bit of the rule fields. Bit can have one of three possibilities; one, zero or don't care (as in the case of wild cards in source/destination address). Accordingly, either the variable, its complement or neither (being omitted) is used in the expression respectively.

For example, consider this rule: Rule: <protocol, source IP, source port, dest IP, dest port> = <any, *.*.*.*, any, 15.32.*.*, 80> the corresponding representation will be

Protocol	Source IP	Source Port
$x_0..x_3$	$x_4..x_{35}$	$x_{36}..x_{51}$
$d \dots d$	$d \dots d$	$d \dots d$

Destination IP	Destination Port
$x_{52}..x_{83}$	$x_{84}..x_{99}$
00001111.00100000.$d \dots d.d \dots d$	0000000001010000

Boolean Expression: Φ only uses variables from x_{52} up to x_{75} plus x_{84} to x_{99}. Therefore, $\Phi = x'_{52} \wedge x'_{53} \wedge x'_{54} \wedge x'_{55} \wedge x_{56} \wedge x_{57} \wedge \ldots \wedge x'_{74} \wedge x_{75} \wedge x'_{76}$....

As we see, 100 variables were reduced to 40 variables. This is common, where rules in policies are normally aggregates, and a smaller percentage of them use specific values for all of its fields. Later, the above mentioned mapping from the firewall rule into the Boolean expression will be used as;

$$\phi = AS(R_i),$$

where ϕ is the Boolean expression representing the address space accepted by Rule i. Or equivalently, it can written as: ϕ = AS (proto, src address, src port, dest address, dest port).

B. Traffic Address Space Segmentation

In order to investigate the behavior as thoroughly as possible while keeping the traffic size at a minimum, it is needed to identify the different possible interactions between rules of the firewall policy in order to generate a small set of packets ($>= 1$) for each different interaction. This can be achieved by intersecting all the traffic address spaces matching the rules of the policy.

Definition 2: A Segment is a subset of the total Traffic Address Space. In a Segment, each of the member elements (*i.e.*, packets, or header tuples) conforms to exactly the same set of policy rules, and no non-member element can conform to this exact set of rules.
In other words, packets belong to the same segment are identical from the point of view of all the rules in the policy.

Each segment is identified by the following information fields:

- AS *(Address Space):* The Boolean expression representing the address space.
- R_{in} *(Rules inside):* Ordered list of rules applying to this space.
- R_{out} *(Rules outside):* Ordered list of rules not applying to this space (*i.e.*, complement of R_{in}; $P - R_{in}$).
- R_{eff} *(Rules effective):* Ordered list of rules that contributed to the final expression of the segment.
- OR *(Owner rule):* The first rule in this list will be taken as the owner of the segment.
- ACT *(Action):* Firewall action to be taken for this space. This is taken as the action of the first rule in the R_{in} list.

Algorithm 1 procedure DoSegmentation (R, defAct, InitDomain)

```
1:  $SEGLIST \leftarrow \Lambda$
2:  AddSegment (InitDomain,$\Lambda$ ,$\Lambda$ , defAct)
3:  **for all** rules: $i$ = 1 to $n$ **do**
4:    **for** segments: $j$ = $SEGLIST.Count$ downto 1 **do**
5:      $S = SEGLIST_j$
6:      $IncSeg \leftarrow S.AS \wedge AS(R_i)$ {Included part of the segment}
7:      $ExcSeg \leftarrow S.AS \sim AS(R_i)$ {Excluded part of the segment}
8:      **if** $IncSeg \neq Seg.AS$ **then** {Segment not contained in the Rule's AS}
9:        **if** $IncSeg \neq \Phi$ **then**
10:         AddSegment ($IncSeg$, $S.R_{in} \cup \{R_i\}$, $S.R_{out}$, $S.R_{eff} \cup \{R_i\}$)
11:         AddSegment ($ExcSeg$, $S.R_{in}$, $S.R_{out} \cup \{R_i\}$, $S.R_{eff} \cup \{R_i\}$)
12:       **else** {there is no intersection between the rule and the segment}
13:         AddSegment ($ExcSeg$, $S.R_{in}$, $S.R_{out} \cup \{R_i\}$, $S.R_{eff} \cup \{R_i\}$)
14:       **end if**
15:     **else** {Segment is inside the Rule's AS}
16:       AddSegment ($IncSeg$, $S.R_{in} \cup \{R_i\}$, $S.R_{out}$, $S.R_{eff}$)
17:     **end if**
18:     SEGLIST.Delete (Segment j) {delete the original segment}
19:   **end for**
20: **end for**
21: **return** SEGLIST
```

The segmentation algorithm constructs a list of segments - (SEGLIST)- according to the rules interaction. Whenever a segment is identified, it calls the AddSegment subroutine to add it. AddSegment takes the policy rules (R_i), the default action of the firewall (named defAct), and the initial domain (named InitDomain) which is the total traffic address space under consideration. If R_{in} is an empty list, the action (ACT) of the segment is set to the default (defAct), otherwise it takes the action used in the first rule (assuming rule priorities are their order in the policy). Similarly, if the R_{in} is non-empty, the first rule is taken as the owner rule (OR).

In Segmentation algorithm(V-B), the first initial segment is initialized, and added to SEGLIST at lines 1 and 2. Then we loop over the rules to impose their effect on all the existing segments. We loop in reversed order over the segments to prevent the newly added segment from being processed in the current iteration. We use three variables just for the sake of readability of the algorithm (i.e., S, IncSeg, ExcSeg). Respectively, they are the currently processed segment, the Included address space between the segment by the Rule, and the excluded space with respect to the rule. Having three cases between the segment and the rule's address space, we either split the segment into included area and excluded area, leave the segment space intact as an excluded segment (if the rule doesn't intersect at all this rule), or leave the segment space unmodified as an included space (if the rule is a superset of the segment).

The conditional adding or modification of segments are necessary to prevent the creation of empty segments. Omitting these two lines guarantees exponential run time in the number of policy rules because there will be a deterministic growth of the number of segments by doubling the number of segments after processing each of the n rules, resulting in a list of 2^n segments.

1) Choosing the Initial Domain:: Until this point we have assumed that the initial domain/space will be the entire possible traffic address space (*i.e.*, corresponding to: the boolean expression: $True$). This is an easy option to start with, but might not be always a practical choice. More efficient choices can be made based on our knowledge (or the amount we would like to involve) of the network. Also, it can differ on wether a robustness test of the firewall is needed as well (*i.e.*, invalid packets are correctly rejected).

Option 1: Using the complete space: This is the simplest space to start with, but it includes packets that we might not be interested in. For example, packets with equal source and destination, or packets having the destination address equals the firewall's incoming interface address. Also, this set makes it possible to select packets that might be invalid with respect to the used protocol. For example, a TCP packet with all control flag bits set, or having source and destination addresses both equal to zero in a normal packet, etc. But on the plus side, this options yields a lower number of segments due to the simplicity of the original domain. In general, this option should not be used except for completeness of all testing scenarios.

Option 2: Using the complete network space, with valid packets only: Just adding the protocol constraints to the space before applying segmentations decreases the size of the space several orders of magnitude (from 2^{148} down to 2^{124}). The packets that result from segments built over this space are guaranteed to be valid, and acceptable by any network device. This is a recommended starting point for general firewall testing.

Option 3: Restriction to network address space: We can start with the traffic address space corresponding to $<$any, *.*.*.*, any, "my network range", any$>$. Starting with this initial segment, is intuitive as it is expected there will be no packets reaching this firewall except those having the firewall protected network's address range (the routers upstream towards the global network will not forward to the firewall except those relevant to my network). However, at least the address spaces of the multicast traffic must be added. We achieve this extension by ORing the AS of the network's range mentioned above, with the AS equivalent to: $<$UDP, *.*.*.*, any, 224.*.*.*/4, any$>$ for multicast traffic.

Option 4: Restriction to a range of source and destination addresses: A logical extension is to test using only packets with a specific range of source and destination addresses. Of course this is typically to be used (along with the previous option) for firewalls that are already installed in a network and need to be tested without disconnecting/reconnecting to the network.

C. Analysis and complexity:

Although the algorithm shown is quite simple, it can yield exponential running time in the general case (*i.e.*, where the Boolean functions are not representing firewall rules, but general expressions). In the case of firewall rules, we have multiple restrictions amongst the Boolean variables involved. For example:

- We cannot specify the i^{th} bit of the source address to be equal to some value, without fixing the values of the $(i-1)$ previous bits.
- The variables representing the ports are either all mentioned in the Boolean expression of the rule or all taken to be "don't care" values. Same applies for the protocol field.

The exponential number of segments occurs in the case where every rule intersects with - at least - a constant ratio of the segments existing from the previous rules. This is a case that is not to be found in a practical firewall policy.

D. Measuring the importance of segments:

It is essential to have a measure for the relevance of each segment, as this leads us to decide how dense the testing should be within a segment. Measuring the importance of the segment (*i.e.*, the probability that any element will be matched incorrectly by the firewall) can depend on (and not limited to) the following factors:

1) *Number of rules in the segment (overlapping rules):* As the number of rules intersecting at the segment increases, so will the importance of the traffic addresses in this segment. The region that is common to many rules can be thought of critical, as handling more overlaps can be harder to the firewall to process.
2) *Number of rules affecting the segment shape (effective/boundary rules):* When a rule intersects non-trivially with a segment splitting it into two non-empty segments, this rule becomes a member of the set of effective rules with respect to this segment. As the number of rules in this list increase, the matching will have a tendency to have more decisions to make for packets within the segment.
3) *The owner rule's (OR) importance:* As the first rule of a segment (owner rule) is considered more important, the whole segment can be judged as having higher tendency of being as important as well, as packets in this segment were designed by the administrator to hit the owner rule.
4) *Cumulative weights of all contributing rules:* Each rule will be having a weight (depending on factors shown below). Also, not all rule weights contribute with the same weight, as we are concerned with the top rules in each segment more than the lower ones. So, there will be a coefficient for each rule that depends on its order in the segment. As a rule become more complex, it can affect the filtering algorithm in its decision process.
5) *Area of the segment:* As the area (*i.e.*, number of traffic address space elements the segment covers) increases, its selectivity (and so its criticalness) decreases, and also our ability to test it exhaustively decreases as well. This means as the area increases the portion of the address space to be tested decreases, consequently its weight. The area can be obtained by checking the number of variables in the expression representing the address space, or the count of different satisfying assignments.

The total testing density of a segment ought to become a function of all of the above shown factors.

$$\begin{aligned}\rho(S) \quad = \quad & w_1|S.R_{in}| + w_2|S.R_{eff}| + \\ & w_3 weight(S.OR) + \\ & w_4 \Sigma_{r \in R_{in}} c(order(r)).weight(r) + \\ & w_5 log_2(\|S.AS\|)^{-1}\end{aligned}$$

Some of these effects can be combined together to simplify the expression:
The third term (summation of rules' weights) is already covering the first two terms, as it sums over all the weights of the rules. So, we can increase the $c(1)$ to take care of the second term. Also, by incrementing the c coefficients in the third term, we can compensate for the first term. The fourth and fifth can be combined as well. Moreover, depending on the suspected implementation that is to be tested, it can be possible to see that taking the logarithm of the address space in another base makes more sense. The resulting expression would be:

$$\rho(S) = w_1 \Sigma_{r \in S.R_{in}} c(order(r)).weight(r)$$

E. Rule Weight:

The segment weight is directly affected by the weights of contributing rules as shown above. Giving each rule in the policy a weight will be easier than that of the segments shown above. The factors affecting the weight of a rule are (and not limited to) the following:

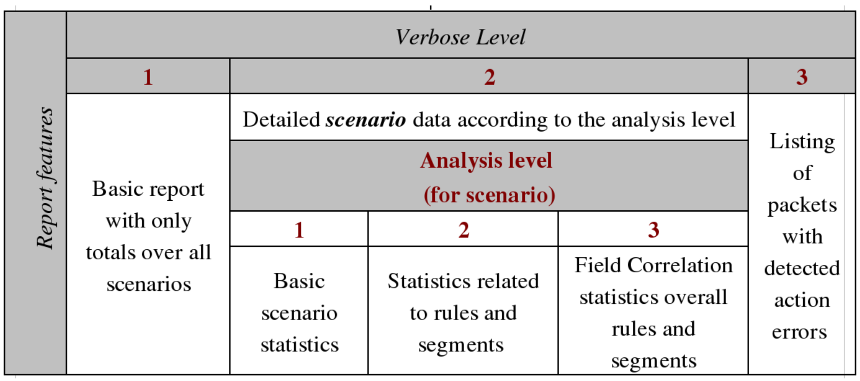
1) *The area the rule covers:* As the rule gets more general, its weight decreases. The more specific the rule the more it is critical to the system and easier to test exhaustively.
2) *Number of fields used:* More options make packet matching harder. Consequently, making errors more possible. This can also be reasoned as follows; every rule is a decision tree with many levels depending on the number of criteria in the rule. Each node in the tree has a chance in being evaluated in error. Therefore, to reach a leaf in such (skewed) tree, the algorithm will face more possible errors as the number of criteria increase.

3) *The number of rules superseding this rule in the policy dependency graph:* As another parameter to the importance of the rule, we take the number of subset rules (or correlated rules as well) that come before it in the policy. The order of a rule in the policy is not the correct value to be taken, as it is not the real order of the rule (as we can move it higher as long as no intersecting rule is encountered).

$$Weight(R_i) = \alpha.\|R_i.AS\| + \beta.\sharp(Higherrules)$$

VI. Reporting

In this section we present the reporting levels of the test results. Two parameters are provided to control the reporting level; Analysis level and Verbose level. The report features are controlled according to the required levels of verbosity and analysis respectively. Figure 8 shows the report features according to the different combination of values for the 2 levels. Note that levels are treated incrementally, (*e.g.*, a verbose level of 3 includes all the features for levels 1 and 2).



	Verbose Level				
	1	**2**			**3**
Report features	Basic report with only totals over all scenarios	Detailed ***scenario*** data according to the analysis level			Listing of packets with detected action errors
		Analysis level (for scenario)			
		1	**2**	**3**	
		Basic scenario statistics	Statistics related to rules and segments	Field Correlation statistics overall rules and segments	

Fig. 8. Report Features

The following results are available from a report with verbose level and analysis level of at least 2.

- *Report summary:* Contains the basic report statistics over all the scenarios included in the run. This includes total number of packets, different errors, and their averages with respect to the number of scenarios
- *Scenario section:* This section is similar to the report summary but is based on the scenario level. Also, it includes the averages and standard deviation of packets and different errors with respect to segment and rules within the scenario
- *Segments subsection:* Contains the summary statistics per each segment in the scenario
- *Rules subsections:* Contains the summary statistics per each rule in the scenario
- *Fields subsections:* Contains correlation figures between errors and the use of specific fields in rules. If the rules with a specific filtering field tends to have matching errors, the strength of this tendency will be shown here.

Higher verbose and analysis levels provide more detailed information into specific errors and failure patterns over rules, segments, and fields. With the knowledge of the underlying filtering algorithm, the product testing cycle can be much more optimized resulting in shorter time to market along with lower development/testing costs.

VII. Evaluation and Results

In order to evaluate the system, we had to investigate the effect of the variation in the main inputs in the performance of the proposed system. The evaluation, as the system design, can be seen in two parts. First one is to evaluate the generation of the policies, to investigate if they cover a wide range of settings, and complexities. The second part is evaluating the packet selection, and how segment-based selection will perform against the basic random packet selection.

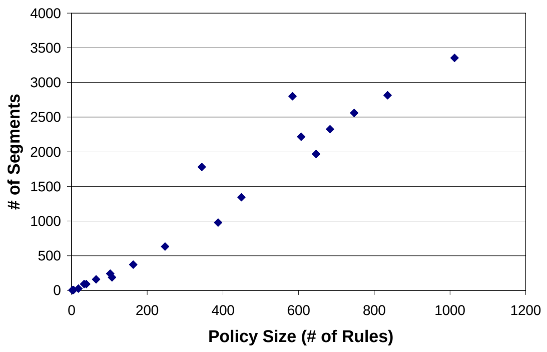


Fig. 9. Number of resulting segments versus the size of policies used. Settings used

A. *Policy Generation Evaluation*

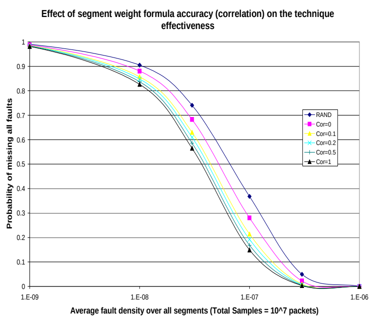
To evaluate a random generator, the basic factor is how random and unpredictable the output is. However, in our case the output size is not large enough for such metric to be of use. Besides, for policies generated to be of more practical use we have to make them more redundant and less random. Thus, we will focus our evaluation on how the generated policies follow the required guidelines and properties provided by the test administrator (*e.g.*, policy size, rule complexity, etc.), and how well the generated policies cover different field values and rule structures (*e.g.*, distinct port values, average rule complexity, use of specific clauses, etc.).

1) Segmentation module: Another point that has to be analyzed is the segmentation step in the testing cycle. As obvious from the definition of segments, they are more in number than the number of rules in the policy. The question is how high can their number be relative to the number of rules. We used a set of generated policies, and performed segmentation on them, and kept the count of the rules and segments. In Fig 9, we can see that the number of segments grows with the number of rules in a policy in a more or less linear fashion. Finding an exact relation between both involves many other factors, like grammar structure, features, and complexity. Also, policy generation options affect the policies in nature. Thus, we can write down a very simple formula that governs the relation to be $\mid Segments(P) \mid = c. \mid P \mid$, where $1 \leq c \leq 5$ and its exact values depends on the above mentioned inputs. For the, more than 250, policies involved in the study it never surpassed this bound. The subset of policies used in Fig 9 were generated from a grammar that forced rules not to have any special filtering clauses (*e.g.*, tos value, ICMP qualifiers, ACK or SYN flags, etc.). This caused the overlapping between rules to be highly probable, and this is the reason we can not find any policies with $\mid Segments(P) \mid \approx \mid P \mid$ (*i.e.*, high proportion of distinct rules). If complex grammars are to be used, the number of segments will drop even further and the values of c might approach unity.

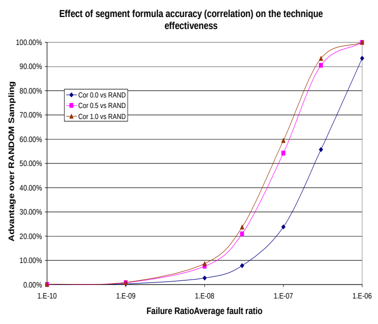
B. *Packet Selection Evaluation*

Comparison will be, mainly, against the random sampler testing mechanism. The operation of the random sampler is as follows: Given an available testing time, calculate the number of possible test packets to be sent, spread these packets uniformly over the whole traffic address space of the investigated firewall. In contrast to the random technique, the proposed technique chooses where to concentrate the samples, and where to allow them to be more sparse; based on the result of the space segmentation. The graphs show the effect of some of the parameters of the system; the effectiveness of the segment weight function in predicting the error rate, the policy style (*i.e.*, , the interrelation between the rules within the policy), and the effect of the segment size skewness.

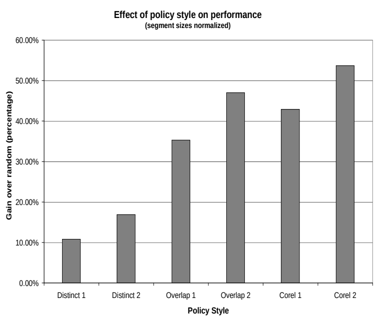
The first (second) graph shows the absolute (relative to random sampler) effect of the effectiveness of the weight function in predicting the probability of error within a segment. As a measure of this effectiveness the correlation between the two vectors (the weight function, and the actual error probability) is taken. It can be seen that any non negative correlation gives a gain over the random sampler. It still gives better results even with zero correlation, this can be attributed to the following; sampling within each segment guarantees a better distribution and ensures that the segments with higher probabilities can not be skipped as in the case of random sampling where whole segments might be skipped. Take into consideration that in these two graphs as well as the second one, we tried



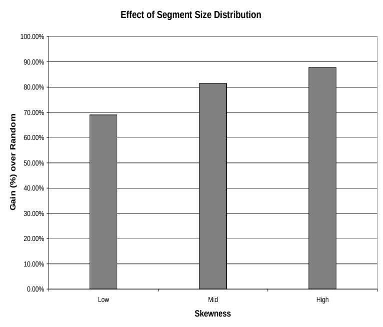
(a) Effect of fault probability prediction accuracy on overall performance (absolute)



(b) Effect of fault probability prediction accuracy on overall performance (relative)



(c) Effect of policy style on the overall performance



(d) Effect of segment size distribution on the overall performance

to be as conservative as we can; all tiny segments were removed to smoothen the effect of high density sampled segments.

It is worth mentioning that this evaluation does not take into consideration the nature of common implementation errors; they mostly cause errors in whole regions rather than have them randomly dispersed in a segment or more. In other words, if there exist an error in the implementation it is highly probable that whole segments (or rules) will be mishandled by the firewall. Thus a single packet per segment will be able to capture the error. In contrast with random sampling that might not even assign a sample from such erroneous segment. By a simple calculation we can find that the probability the random sampler will miss hitting an erroneous segment is close to 100% even after using millions of packets (See Table II). We can write down the probability that the random sampler will miss the segment as follows: $P(miss) = (1 - 2^{s-S})^N$, where s and S are the sizes of the segment and total space in bits, and N is the number of packets to use.

Secondly, the policy style is investigated in the third graph. After removing very high weighted segments, as well as tiny segments (those below a threshold are tested exhaustively, thus causing our technique to be superior over the random sampler and hiding the effect of the correlation, style and any other parameters), all styles behave quite well. Of course there is a general tendency that those policies with high interaction and/or many very specific rules (e.g., where all tuples are specified) would give better performance for our technique rather than the naive

Total Space (Bits)	Error Segment (bits)	Prob of hitting w' one Pkt	Prob of missing all faults
Using 10^6 Packets			
104	0	4.9303 E-32	1
104	32	2.1175 E-22	1
104	64	9.0949 E-13	0.999990
58	0	3.4694 E-18	1
58	32	1.4901 E-08	0.985209
58	40	3.8146 E-06	0.022044
Using 10^9 Packets			
104	0	4.9303 E-32	1
104	32	2.1175 E-22	1
104	64	9.0949 E-13	0.99909
104	96	0.0039	0
58	0	3.4694 E-18	1
58	24	5.8207 E-11	0.94345
58	28	9.31323E-10	0.39403
58	32	1.4901 E-08	3.3768 E-07

TABLE II

PROBABILITY THE RANDOM SAMPLER WILL MISS A SEGMENT. THE SIZE OF THE TOTAL SPACE IS GIVEN BY THE NUMBER OF FREE BITS IN ITS BOOLEAN EXPRESSION (*i.e.*, LOG(SPACE SIZE)). SIMILARLY, THE ERRONEOUS SEGMENT'S SIZE IS GIVEN.

random sampling counterparts.

Thirdly, to include the effect of small segments, the fourth graph shows how including small segments and those with very high weight can render the segmentation-based sampling a superior technique in investigating the firewall performance in ratio to the random sample. Gain higher than 99% was attained in some of the tested policies. As a conclusion, the segmentation based technique gives orders of magnitude of better results over random sampling when utilized to use all the features in the tested firewall/policy (e.g., exhaustive testing of very small segments, high sampling for high weight/moderate weight segments, ...).

VIII. CONCLUSION AND FUTURE WORK

In this paper, we present an automated framework for testing the filtering implementation of firewalls. Our technique avoids the exhaustive testing via exploiting the interaction of the rules within the firewall policy. Using this information, a carefully selective generation of testing packets is achieved that will result in a small probability of missing all fault locations. Also, a random policy generator was developed to generate customized policy configuration to be enforced by the firewall device under test. The evaluation results show that the correlation between the assumed and the actual probability of error can vary while maintaining better results than random sampling. Higher success probabilities have been achieved in many cases; 40-60% is a common gain over the sample policies studied. Also, several policy styles have been tested, and the new approach proved to be superior in all cases. The policy generator was shown to generate policies with a wide range of customizable properties accurately.

Currently, research is in progress to enhance the policy generator to incorporate more options, and capabilities to generate human like policies. Also, taking into consideration that policies should be as orthogonal in their relation with the filtering algorithm as possible, renders the selection of random policies a really hard problem. Studying the segmentation behavior for several policy styles needs further investigation. Also, there are several well known families of filtering techniques; tweaking the system and weight functions in order to target each one of these families is also under investigation.

ACKNOWLEDGEMENT

This work was funded by Cisco as part of INSPEC project.

REFERENCES

[1] A. Wool A. Mayer and E. Ziskind. Fang: A Firewall Analysis Engine. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy (S&P'2000)*, pages 85–97, August 2000.

[2] W. Richards Adrion, Martha A. Branstad, and John C. Cherniavsky. Validation, Verification, and Testing of Computer Software. *ACM Computer Survey*, 14(2):159–192, 1982.

[3] K. Al-Tawil and I. Al-Kaltham. Evaluation and Testing of Internet Firewalls. *International Journal of Network Management*, 9(3):135–149, 1999.

[4] B. Beizer. *Software testing techniques (2nd ed.)*. Van Nostrand Reinhold Co., New York, NY, USA, 1990.

[5] B. Beizer. *Black-Box Testing Techniques for Functional Testing of Software and Systems*. Wiley-VCH, 1995.

[6] R. Haeni. Firewall penetration testing. Technical report, The George Washington University Cyberspace Policy Institute, 2033 K St, Suite 340N, Washington, DC, 20006, US, January 1997.

[7] J. Jürjens and G. Wimmel. Specification-Based Testing of Firewalls. In *Proceedings of the 4th International Conference on Perspectives of System Informatics (PSI'02)*, pages 308–316, 2001.

[8] Microsoft. Network devices testing guidance. Microsoft Technet, March 2005. http://www.microsoft.com/technet/itsolutions/wssra/raguide/NetworkDevices/igndbg_4.mspx.

[9] M .Ranum. On the topic of firewall testing.

[10] G. Vigna. A formal model for firewall testing.

[11] J. Walsh. Firewall testing: An in depth analysis. ICSA Labs Techncial report, June 2004. www.icsalabs.com/icsa/docs/html/communities/firewalls/pdf/fwwhitepaper.pdf.

[12] A. Wool. Architecting the Lumeta Firewall Analyzer. In *Proceedings of the 10th USENIX Security Symposium*, August 2001.