Automated Pseudo-live Testing of Firewall Configuration Enforcement

Ehab Al-Shaer, Adel El-Atawy, Taghrid Samak
School of Computer Science, Telecommunication, and Information Systems
DePaul University
Chicago, Illinois 60604
Email: {ehab, aelatawy, taghrid}@cs.depaul.edu

Abstract—Network security devices such as firewalls and intrusion detection systems are constantly updated in their implementation to accommodate new features, performance standards and to utilize new hardware optimization. Reliable, yet practical, testing techniques for validating the configuration enforcement after every new software and firmware update become necessary to assure correct configuration realization. Generating random traffic to test the firewall configuration enforcement is not only inaccurate but also impractical as it requires an infeasible number of test cases for a reasonable testing coverage. In addition, in most cases the policies used during testing are manually generated or have limited configuration profiles.

We present a framework for automatic testing of the firewall configuration enforcement using efficient and flexible policy and traffic generation. In a typical test session, a large set of different policies are generated based on the access-control list (ACL) grammar and according to custom profiles. Test packets are generated to particularly consider critical segments of the tested policies and to achieve high coverage of the testing space. We also describe our implementation of a fully-automated framework, which includes ACL grammar modeling, the policy generation, test cases generation, capturing and analyzing firewall output, and creating detailed test reports. Our evaluation results show that our security configuration testing is not only achievable but it also offers high coverage with significant degree of confidence.

Index Terms—security configuration testing, firewall testing, policy enforcement validation, security evaluation, automated security analysis, policy generation, network security.

I. INTRODUCTION

A valid implementation of network security configuration is an integral part in the successful operation of any security device. Firewalls are the cornerstone of network security that operate based on access control list configurations. Due to the persistent effort to enhance and optimize firewall protection and performance, firewalls undergo continuous modification. Deploying new optimized filtering algorithms, adding new features into the access-control list (ACL), and extending the syntax and semantics of firewall polices are examples of routine updates in firewall software. This increases the potential of software bugs that create invalid implementation of filtering, causing discrepancies between the policy configuration (ACL) and the actual actions enforced by firewalls. Therefore, extensive and thorough configuration conformance testing is necessary after each new firewall software release for any vendor in order to uncover any errors in the filtering implementation and guarantee product quality assurance.

Firewall implementation testing requires two phases: (1) generating policies (i.e., ACL) with different configurations such as rule complexity, rule interaction, filtering criteria, rule space, etc. and (2) generating packets to test the enforcement (or implementation) of the device under test (DUT) using these policies. Our presented framework addresses both problems by developing flexible policy generation techniques applicable to any ACL grammar specification, and efficient traffic generation techniques using policy segmentation.

Generating firewall policies that are effective in automated configuration testing is a challenging problem in its own. First, the policy generator should be applicable to different firewalls with different policy specification languages. Second, the generated policies should cover the rule configuration effectively considering field values, rule complexities and rule inter-relations (i.e., overlapping, super/subsets, etc), degree of interaction, rule space and others. We address the first challenge by allowing users to define any ACL policy syntax using a customizable form of augmented context-free grammar. The second challenge was addressed by generating the rules based on independent representation (i.e., a push down automaton accepting the grammar) driven by policy profile distribution. Thus, this allows to separate the logic of policy generation from the policy specification.

Testing the firewall by exhaustively injecting all possible packets into the firewall is not feasible due to the huge number of packets needed in this case, even if restricted ranges based on domain address are used. Restricting the domain by confining packets to realistic values and tuning the encoding will reduce the required testing time from about $4 \times 10^{13}$ years for the complete space, to 4.5 years (using one billion packets per second) [10]. 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. For example, if web traffic (tcp, port 80) is being mishandled by a firewall, then one million random packets will miss the error with probability around 94% (which is $(1 - 1/2^{8+16})^{10^6}$). We will show that our system is almost guaranteed to find this bug with a much lower number of test packets. Therefore, we propose a novel testing technique for...
ACL devices based on fine grain policy segmentation (called “primitive segments”) and smart packet selection in order to achieve enhanced coverage of all the traffic space. A main advantage of our system over other sampling-based testing techniques is our exhaustive-nature sampling of decision paths rather than testing space.

The presented framework is composed of several subcomponents: policy generation, traffic selection/injection, post-test analysis/reporting, as well as other supporting components. The system prototype has been implemented and tested by one of the major network device vendors using large number of various polices sizes and firewall device operating systems (i.e., several releases of IOS, PIX, etc). The system was successful in discovering various kinds of undisclosed bugs in beta-versions of the firewall OSs. Some bugs were unlisted, but were discovered by the vendor previously, and others were introduced for the purpose of testing the power of our system.

In the next section, an overview of related work is provided. Section III discusses the system framework and its different modules. The policy generation and test data generation are discussed in sections IV and V, respectively. In sections VI and VII, system implementation and evaluation are presented followed by conclusion and future work.

II. RELATED WORK

A concrete and valid security configuration is essential to the success of any IT-based environment. Validating the correctness of configurations (e.g., firewall policies) based solely on their definition [3], [13] is definitely not enough. Guaranteeing correct realization of the configuration into a valid operational form is necessary for complete assurance of quality. Therefore, testing the implementation itself is an integral part of the overall process. Product testing is often considered the most expensive and time consuming phase in any production line. In black-box testing, the samples required to perform exhaustive testing can be impractically large, a subset of the input data is selected statistically [2], [5]. Firewall errors can be categorized into four error types: security, implementation, policy and capability errors [21]. Each error type has its own set of detection techniques. Generally, we can categorize these testing techniques into theoretical approaches [14], [19], [22] and practical approaches [4], [12], [21].

In the theoretical approach, the testing is based on a formal model of the firewall and the surrounding network. In [19], a firewall testing methodology is proposed based on a formal model of networks that allows the test engineer to model the network environment of the firewall and verify that the topology of the network provides the needed protection. Also, in [14], a CASE tool is used to model the firewall and the surrounding network then test cases are generated to test the firewall against a list of security vulnerabilities. Others, like [22] and [1] introduced a firewall analyzer tool that is capable of analyzing firewall configuration parameters from real environments by simulating firewall behavior. More recently, other approaches were presented [13], [3] and [23] to model and analyze firewall behavior based on given configuration. All these approaches attempt to verify the correctness of firewall policy but not the implementation.

In the practical approach: most of the previous work provides methodologies to perform penetration testing against the firewall. Ranum [16] 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 those 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 [4] 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 [12] 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 tested against a pre-defined set of firewall violations (i.e., firewall logging capability, firewall security vulnerability, firewall policy, etc.) that are described by Walsh in [21] and the one that passes their test gets certified.

Generating network traffic that is variable enough to imitate natural behavior is not a simple task [11]. The aim of several previous trials [7], [17], [20] was to generate traffic that follows closely Internet traffic. In [7], HTTP traffic was the specific target. In [20], a tool, Swing, was developed that creates models of users, applications, and protocols to be followed by the generator in order to represent the immense complexity of background traffic. Harpoon, the tool presented in [17] extracts behavior from router logs and Netflow archives and generates traffic at the IP flow level. The goal was to use the generated traces for performance analysis and benchmarking rather than case coverage and testing.

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 and the desired configuration correctly or not. The closest work in the area of testing and evaluating firewall devices is ClassBench [18]. This tool is capable of generating rule sets that follow a probability distribution guidelines provided by the user for each of the requested fields. Also, traffic generation is provided in the tool with controllable locality of packets selected. However, the policy/rule generation in this work does not consider rule complexity and fields’ interactions features. In addition, this tool lacks flexible and customizable model of policy grammar for policy generation, and its traffic generation does not guarantee an efficient coverage of testing space.

III. SYSTEM FRAMEWORK

An overview of the system shows three components: the Engine, the Spy, and the User Interface. The Engine is the

1 Although the discovered bugs were in pre-release and development versions, we cannot discuss the details of these bugs due to non-disclosure agreements.
core component and it handles most of the processing. 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 an architectural point of view, 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). As the test starts, the engine generates a policy, analyzes the policy for traffic selection and loads it into the firewall. The digested policy in the form of segments (will be discussed in section V-B) will be used to generate packets to be injected into the firewall. The outcome of the firewall will be monitored by the Spy process in the form of permit-deny map. Afterwards, the map along with the policy will be analyzed and a report will be generated. This cycle can be repeated as the administrator wishes to reduce the chances of false negatives. 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 (Policy Grammar) Parser:** The parser reads the grammar specifications 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. The grammar is described in Sec IV and detailed examples can be found in [9], [10].

- **Policy Generation Module:** This module generates policies according to a set of parameters (including grammar specification) as provided by the user. Optionally, users can load policies manually, thus overriding this module. The output of this module is a policy in plain text that follows the firewall syntax.

- **Policy Parsing, Checking and Compiling:** It parses a policy, and compiles it into an internal representation of constraints on packet header field bits.

- **Segmentation Module:** This component analyzes the policy and generates a list of segments that represents the whole policy. See section V-B for details.

- **Segment Weight Analyzer:** Segments are analyzed and each is assigned a weight based on a number of factors (e.g., number of intersecting rules, segment space, etc). The segment weight is a measure of its criticality and the chance of being mishandled by the firewall.

- **Test Packet Generator:** Packets are generated and distributed in proportion to the segment weight. Each test packet carries test case information that includes a packet sequence number, the test session ID, and a signature to mark it as a test packet.

- **Spy: firewall monitoring tool:** The spy module is a separate application that monitors the output of the firewall, and records all permitted packets. This list of packets is compared against the expected list received from the Engine and the discrepancies are reported back for analysis. More details are shown in Section VI.

- **Post-Test Analyzer** The information collected by the spy is 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.

---

The fact that all policy grammars do not contain repeating components makes them a weaker version of context-free-grammars, and can be represented via a FSA.
IV. POLICY GENERATION

In this section, we represent 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. The grammar defining the firewall is parsed, then provided to the generator that produces customized policies. The parser/compiler then converts those policies to an intermediate structure ready for analysis.

A. Policy Grammar

Defining the ACL syntax can be a lengthy but straightforward task. However, specifying the semantics of the different clauses in the grammar (i.e., what these tokens and clauses mean) is a different problem. Embedded annotations are needed to describe these semantics to the grammar parser and the policy generator afterwards. For example, having an IP clause in the syntax does not clarify whether it is the subnet address, or the mask and there is no indication on which IP field it should be mapped to: source or destination.

Also, keywords in the syntax should be mapped to either a value (e.g., “tcp”=6 protocol, “domain”=53 port value, etc.), or a specific operation (e.g., “eq”, “gt” for port values, and “permit/deny” for rule action, etc.). Moreover, cross-field constraints are hard to define with a direct definition (e.g., “permit/deny” for rule action, etc.). Moreover, cross-field constraints are hard to define with a direct definition (e.g., “http” port with “udp” should not be allowed).

From a theoretical point of view, adding the annotations needed for the above mentioned tweaks will not result in a more powerful grammar. This observation is necessary in order to guarantee that a simple FSA (Finite State Automaton) can recognize this grammar successfully (i.e., accepts any policy following this grammar). Figure 4 shows the grammar corresponding to the extended IOS access list [8]. The complete grammar can be found in [9], [10].

In our implementation, all annotations are prefixed with a backslash for parsing purposes. Special keywords/operations in the grammar are:

- \FieldID(n): A pre-assigned field identifier (n) to be used for the following sections and in conditional statements to identify the physical packet field.
- \num(n1,n2): An integer range from n1 to n2.
- \V(x): The value of the current token is x. For example, “tcp”\V(6) means the string “tcp” has the value of 6.
- IPvalue and IPmask: Special handling for IP and mask.
- \Trans: This is required to specify extra handling of data in the rule. For example, conditions with inequalities over port values, and the subnet mask effect on IP addresses. 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 defining the context of the symbol (i.e., IP, port, protocol), the data type of the last parameter, and the action to apply.
- \Lookup: Used to define a list of constants retrieved from files of name-value pairs. For example, port and protocol names (instead of listing them via the “\V” keyword).
- \Cond: Some rules in the grammar are only valid if a specific value is assigned to one of the fields. Mostly, this takes place when clauses depend on the protocol value. It is associated with a non-terminal, and 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 finite state automata of the grammar. The graph has a starting state and a unique final accepting state. In between those nodes, the graph is built 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 explains how this graph is used for policy generation.

Each non-terminal in the grammar is expanded into a subgraph between two nodes. Outgoing edges from a graph node represents the different options for the syntax. Graph
edges store information about field IDs, dependency conditions between fields, and other annotations provided in the grammar. Each edge contains the necessary information to decide a part of the syntax: \( <s,v,f,c,probability> \). The first item is the string text to be matched, followed by the value to be assigned if matched. The third item is the concerned packet field. The fourth value specifies any cross-field dependencies (which is itself a list of field-value pairs). The last item is a collection of probability distribution guidelines for this field (see section IV for details). Figure 5 shows the portion of the graph corresponding to the IP subgraph.

Policy rules are parsed using this graph. The process begins at the start state, then each encountered token is matched against outgoing edges/links from the current node. Reaching the final state successfully results in accepting the rule.

C. The Generation Process

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 and rules are built according to the probabilities specified at each link. In our generation, two modes are available for setting the probability values: 1) static-random and 2) informed modes.

1) Static-Random Mode: This mode is the default for the system, where some general information is provided to the generator. The probabilities on each link are preset based on the intuition and common administration practices. For generating a policy, there is a large set of customizable parameters that control the configuration of a policy such as (1) policy size: the average number of rules that will be generated for each policy, (2) average rule complexity: the probability for an optional field to be used; the higher the value the more complex the rule will be, and (3) probability of accept: the probability of choosing the action of the rule to be permit. Another set of parameters are used to control the way the generator selects values for different fields. For instance, policy generation can be optionally configured to favor the lower range of port values or exhaust a range of values, whenever possible, for specific field(s).

2) Informed Mode: This generation mode is more dependent on the domain, and prior policies defined by the administrator. A set of policies is provided to the system to learn the probabilities for the graph links. This way, the generation is more likely to produce policies following the administrator’s general behavior. The learning is a two-level process. The first level learns the field values probabilities and inter-field relations. The second level is taking into consideration changing probability values within policy parts. In our analysis, we used real-life anonymized policies provided by CISCO, with over 50,000 total number of rules.

   a) Level-I: For each given policy, the policy graph is simply traversed as if the policy is being parsed for syntax correctness as described in section IV-B. A counter maintained at each link of the graph, is incremented after each visit to this edge. After all policies are parsed, the link weights for each node are normalized to reflect the probability. On a quick view, this can be considered as the first-order statistics for field values. On the other hand, multiple edges are being updated during the traversal which reflects fields interactions, in contrast to prior work that estimated single field probabilities [18]. For example, some parts of the graph are not visited unless specific protocol is used. The learning will capture this relation without the need of explicitly stating the interaction.

   b) Level-II: It is commonly observed that early rules tend to have specific values in their fields, while later rules in the policy tend to use general values like wild cards. This is also observed in the correlation between the position of the rule within the policy and link weights for many fields. To learn this property, probabilities for different parts of the policy are learned independently. More specifically, each policy is partitioned into a number of rule groups (consecutive rules). Probabilities for each partition are then learned independently, following the same procedure from level-I. If two consecutive groups have similar distribution they are then merged. The question now is where to put the partition boundary (i.e., how many rules in each partition). In this process multiple graphs will be generated corresponding to different policy partitions, with their order maintained. The general step in this procedure will have a current graph \( G_c \) and an incoming rule \( R_i \). Considering \( R_i \) to belong to \( G_c \), will result in \( G_c' \) with new statistics. We say that \( R_i \in G_c \) iff \( G_c' \gg G_c < T \) where \( \gg \) calculates the discrepancy between the two graphs. This means that, \( R_i \) does not contribute to \( G_c \) unless it matches the overall profile of the graph within a certain threshold \( T \). If the condition is not satisfied then a new graph is generated, and the partition is defined.

At the final stage we will have a number of graphs corresponding to policy partitions along with the average proportion of the policy rules contributing to each graph. Policies are then generated by traversing the graphs in order, and generating the corresponding number of rules.

V. SMART Traffic Generation

In this section, we describe the method used for generating test traffic. The main goal is to generate the least amount of traffic that is needed to test all the possible decision paths for the given firewall configuration.

We define \( p \) as a packet that is identified by a \( d \)-tuple of all fields \( F_i \) used to describe traffic: this is usually composed of \( <\text{protocol}, \text{source address}, \text{source port}, \text{destination address}, \text{destination port}> \) plus any extra fields as TCP flags, icmp codes, etc. Now, we can define the whole traffic space as follows:

**Definition 1:** Traffic Space \( TS \) is the space whose elements are packets \( p \) representing all different combinations of field values. In other words, \( TS = \{ p =< f_1, \ldots, f_d > : f_i \in F_i \} \)

Our goal is to create test cases/samples intelligently in the traffic space such that filtering decision paths are efficiently covered. This will be the focus of our discussion in the following subsections.

A. Firewall Rules Representation

We will represent rules and traffic subspaces as Boolean expressions. All packets belonging to a certain rule must
satisfy its boolean expression. We chose this representation over others (e.g., geometric, set-based, etc) as it facilitates operations like exclusion and intersection of policy rules’ domains while encapsulating all special case handling. For example, finding packets that will match a specific rule but not another will require a single conjunction (and a cheap negation) and will result in a single expression, while the geometric representation can have a worst case of $2^d$ hyper-rectangles (in a $d$-dimensional space) and the set-based approach needs some sort of explicit representation which is not feasible in our application domain. Moreover, we use ordered binary decision diagrams (OBDDs) to implement the binary expressions and its operations, which simplifies the logic and provides excellent performance while handling quite complex expressions (see section VI for more details).

In the following section, we will show how such representation will simplify test case generation and analysis. The rule expression is created by assigning a variable to each bit of the rule fields. Bits 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: <proto, src IP, src port, dst IP, dst port> = <tcp, *, *, *, any, 15.32.**, 80> the corresponding representation will be

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Source IP</th>
<th>Source port</th>
</tr>
</thead>
<tbody>
<tr>
<td>x7..x0</td>
<td>x39..x38</td>
<td>x55..x40</td>
</tr>
<tr>
<td>00000110</td>
<td>d...d</td>
<td>d...d</td>
</tr>
</tbody>
</table>

The Boolean expression $\Phi$ will be equal to $x_0 x_1 x_2 x_3 \ldots \land x_7 x_8 x_9 x_{10} x_{11} x_{12} x_{13} x_{14} x_{15} x_{16} x_{17} x_{18} \ldots$. As we see, from 104 variables only 40 variables were used. This is far from being uncommon as 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 denoted by; $\phi = AS(R_i)$, where $\phi$ is the Boolean expression resulting from the mapping function $AS(\cdot) : R \mapsto B$ ($AS$ stands for address space, $R$ is the set of $p$ satisfying a rule, and $B = \{b : 2^n\}$ and $n$ is the number of variables used in the representation).

### B. Traffic Space Segmentation

To investigate the behavior as thoroughly as possible while keeping the traffic size at a minimum, we need to identify the different interactions between rules of the firewall policy in order to generate packets that targets 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 space such that any two traffic elements (i.e., packets, or header tuples) are members in the same segment iff they match exactly the same set of policy rules in the policy, and the opposite is true (i.e., no non-member elements can conform to this exact set of rules).

![Algorithm 1 DoSegmentation (R, defAct, InitDomain)](attachment:image)

Formally, for any two packets ($p_i$ and $p_j$) and any segment $S$ and rule $R_k$:

$$[p_i \in S \land p_j \in S] \rightarrow [\forall R_k : (p_i \land R_k) \leftrightarrow (p_j \land R_k)]$$

Note that $p_i \land R_k$ iff the $p_i$ matches $R_k$. In other words, packets belonging to the same segment are identical from the point of view of all the rules in the policy.

Each segment is represented as a tuple: $<AS_{\text{in}}, R_{\text{in}}, R_{\text{eff}}, OR, ACT>$, where $AS$ (address space) is a Boolean expression representing the space of the segment, $R_{\text{in}}$ (rules inside) is an ordered list of rules applying to this space, and similarly $R_{\text{out}}$ (rules outside) is the ordered list of rules not applying to this space (i.e., complement of $R_{\text{in}}$; $P - R_{\text{in}}$). $R_{\text{eff}}$ (rules effective) is the ordered list of rules that contributed to the construction of the final expression of the segment (i.e., boundary rules). The first rule in $R_{\text{in}}$ is designated as the **Owner rule**, OR. Finally, $ACT$ is the action for this space. This is deduced from the action of the owner rule.

The segmentation algorithm constructs a list of segments - $SEGLIST$ - according to the rules interaction. Whenever a segment is identified, it is added by the $AddSeg$ subroutine. $AddSeg$ takes as input the segment expression, intersecting policy rules ($R_{\text{in}}$), and the list of effective rules ($R_{\text{eff}}$). If $R_{\text{in}}$ 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 policy order dictates rule priority). Similarly, if the $R_{\text{in}}$ is non-empty, the first rule is taken as the owner rule (OR).

In the segmentation algorithm, Alg. 1, the first segment is initialized, and added to $SEGLIST$. Then, a simple loop is used to reflect the effect of all rules in order. Three variables are used for the sake of readability of the algorithm (i.e., $S$, $IncSeg$, $ExcSep$). Respectively, they are the currently processed segment, the included space between the segment and the Rule, and the excluded space with respect to the rule. Having three cases between the segment and the rule’s space, we either split the segment into included area and excluded area, leave the segment space intact as an excluded segment (if the rule and segment do not intersect), or leave the segment space unmodified as an included space (if the rule
is a superset of the segment). The conditional addition/update of segments is necessary to avoid creating empty segments. These two lines are important to avoid exponential run time in the number of policy rules as the number of segments will be deterministically growing by doubling their number after processing each of the \( n \) rules, resulting in a list of \( 2^n \) segments. The number of segments generated for policies of different sizes can be seen in Fig 8.

After discussing the segments definition and their extraction algorithm, it is important to show formally an important property they guarantee: covering different rule-rule interaction.

**Lemma 1:** Algorithm 1 extracts the list of all segments for a firewall policy.

**Proof:** We will use induction to prove the Lemma. For an empty policy \(( n = 0 )\), there exist a single segment that is initialized in the beginning of the algorithm (line 2) and corresponds to the default rule (default-deny rule). Let us assume after processing \( n \) rules, a list of \( S \) segments is produced. If a new rule \( n + 1 \) is added, then there are three possibilities: either the added rule is disjoint, overlapping or subset of a segment. In case of overlapping, the new rule \( n + 1 \) will split any intersecting segment into two segments. The code condition in Line 6-8 \(( r \land s \neq false)\) guarantee this property. All three cases are considered and the resulting overlapping and non-overlapping segments are added to the total segment list (line 10 and 11 for overlapping, 13 for disjoint and 16 for subset case). This produces all possible segments for \( n + 1 \) rules. Therefore, processing any \( m \) rules by this algorithm will generate segments that cover all interactions (decision paths) of a list of rules.

The next theorem will show the relation between testing based on segments and potential firewall bugs due to rules interactions.

**Theorem 1:** Given a set of segments, \( S_1 .. S_n \), that represent the policy \( P \), if there is a specific rule dependency order, or interaction that produces incorrect implementation of the policy, then this can be discovered by testing at least one of these segments.

**Proof:** We will prove this by contradiction. Based on Lemma 1, \( S_1 .. S_n \) is a correct and complete set of policy \( P \). In other words, \( P = \bigcup_{i=1, n} S_i \). Let us assume that there is a specific rule ordering scenario of overcalling rules that yield incorrect firewall behavior (action) for all traffic matching the corresponding overlapping area \(( f )\), but this bug (incorrect action) can not be detected when testing any of the segments. This means that either \( S_1 .. S_n \) is incomplete set or \( f \notin P \). Both conditions are contradictions with our original stating facts. Therefore, this scenario of inconsistency between policy specification and implementation must be detected by testing these segments.

**C. Primitive Segments Extraction**

Segments can be very irregular in shape, and are generally composed of multiple rectangular areas that coalesce together forming the overall segment. As segments are represented as a Boolean expression, primitive segments \(( PS \) are the mutually exclusive minterms that constitute the overall expression. For example, if a segment \( S = x \lor y \), then there will be 2 \( PS: PS_1 = x, PS_2 = \neg x.y \). In our implementation, primitive segments of a segment \( S \) are stored in a list \( PSLIST \).

Formally,

\[
S.AS = \bigvee_{i=1}^{\vert S.PSLIST \vert} AS(S.PS_i),
\]

s.t. \( AS(S.PS_i) \land AS(S.PS_j) = false, \forall i \neq j \)

Figure 6 shows that segment \( S_3 \) can be divided into four \( PS's: PS_{3.1} \) to \( PS_{3.4} \).

In order to maximize the test coverage for any firewall implementation, we attempt will make sure that the samples selected from each segment hit as many primitive segments as possible. However, the number of minterms \(( i.e., PS's) \) per segment can be excessively large. Thus, a careful extraction of the minterms should be used to provide efficient as well as scalable testing. We first limit the total number of \( PS's \) used by uniformly allocating a maximum number of \( PS \) for every segment as follows: \( PS_{thr} = PS_{max}/\vert SEGLIST \vert \) where \( PS_{max} \) is a limit on the total number of \( PS's \) that will be used in testing. In this case, if a segment consists of high number of \( PS's \), the segment will be represented as follows:

\[
S.AS = \bigvee_{i=1}^{n-1} AS(S.PS_i) \lor S.AS',
\]

s.t. \( AS(S.PS_i) \land AS(S.PS_j) = false, \forall i \neq j \) and \( S.AS' \land AS(S.PS_i) = false, \forall i \)

where \( S.AS' \) represents the residual space after extracting a group of \( n - 1 \) \( PS's \). Test traffic samples are extracted from each \( PS \) (including the residual space) by finding satisfying assignments \([6], [15]\).

In algorithm 2, we iterate through all segments and extract \( PS's \) until the threshold is achieved. If after a full iteration, we did not reach the limit yet, we increase the threshold per segment accordingly and loop again over segments that were not sufficiently consumed. This iteration continues until \( PS_{max} \) is reached, or all segments have been depleted. A segment is considered depleted if it meets one of three conditions: (1) \( PS's \)’s extracted completely covers its expression, (2) number of \( PS's \) is larger than a first threshold \( PS_1 \) and the
Algorithm 2 ExtractPS (SEGLIST, PS\textsubscript{max}, IterMax, SegCoverage)

1: \{SEGLIST: Original List of Segments\}
2: \{PS\textsubscript{max}: Maximum overall PS count\}
3: \{IterMax: maximum number of iterations over SEGLIST\}
4: \{SegCoverage: percentage of segment area required for depletion\}
5: for all segments: \(i = 1 \ldots \text{SEGLIST}.\text{Count}\) do
6: \text{SEGLIST, PS\textsubscript{LIST} ← Λ}
7: end for
8: \text{SEGDEPLETED ← Λ}
9: \text{Iter = 0, PSCount = 0, curSeg = 0}
10: \text{PS\textsubscript{thr1} = PS\textsubscript{max}/|\text{SEGLIST}|, PS\textsubscript{thr2} = PS\textsubscript{thr1}/2}
11: while \((\text{Iter < IterMax}) \land (\text{PSCount < PS\textsubscript{max}})\) do
12: \text{S = SEGLIST}_{\text{curSeg}} \{shorthand for clarity\}
13: \text{ps\_Term = getSATOne(S, \text{AS})}
14: \text{S.AS = S.AS - ps\_Term}
15: \text{S.ExtractedArea = Area(ps\_Term)}
16: \text{S.PS\textsubscript{LIST} ← ps\_Term}
17: if \((S.AS = \text{false}) \lor [(S.PS\textsubscript{LIST} > PS\textsubscript{thr1}) \land (S.Area - S.ExtractedArea) < S.Area < \alpha] \lor (S.PS\textsubscript{LIST} > PS\textsubscript{thr2})\) then
18: \text{SEGLIST} \leftarrow \text{S}
19: \text{SEGDEPLETED} \leftarrow S
20: \text{else}
21: \text{curSeg = (curSeg + 1) mod SEGLIST.Count}
22: \text{if curSeg = 0 then} \{A complete iteration over segments is completed\}
23: \text{Iter} + = 1
24: \text{PS\textsubscript{thr1} = PS\textsubscript{thr1} + PS\textsubscript{thr1}}
25: \text{PS\textsubscript{thr2} = PS\textsubscript{thr2}}
26: \text{end if}
27: \text{end if}
28: \text{PSCount} + = 1
29: end while

coverage in terms of possible satisfying assignments is larger than a threshold \(\alpha\), or (3) number of PS’s extracted reached a higher threshold \(PS_2\). For example, if \(PS_1 = 8, PS_2 = 16\), and \(\alpha = 0.75\) then a segment is depleted if using less than 8 PS’s if it was completely covered, 0.75 of its area was covered using 8 to 16 PS’s, or 16 PS’s limit was reached regardless of coverage. After each iteration over all segments, the thresholds \(P_1\) and \(P_2\) are increased to reflect the remaining PS quota.

D. Measuring the importance of segments

It is essential to have a measure for the relevance of each segment, as this helps us decide how dense the testing should be within this segment. The importance of the segment (or the probability that a packet will be matched in error) depends on (and not limited to) the following factors:

1) Number of overlapping rules in the segment: The region that is common to many rules can be thought of as critical, as handling more overlaps can be harder for the firewall to process.
2) Number of rules affecting the segment shape (effective/boundary rules): A rule that intersects non-trivially with a segment is a member of the set of effective rules of both sub-segments. As the size of this set increases, packets within the segment will need more evaluations to be matched.
3) The owner rule’s (OR) importance: If the OR is harder to implement by the firewall, the whole segment is considered more error prone (as packets in this segment should hit the OR first).
4) Cumulative weights of all contributing rules: The more complex the rule the more it affects its segments. Also, the higher the rule in a segment’s \(R_{in}\) list the more important it is for this segment. Therefore, we add an order-dependent coefficient for scaling the weight of each rule.

5) Area of the segment: The more constraints on a segment definition, the smaller its area. Thus, the smaller the area the harder it is on the firewall to reach the correct conclusion. In Section VI, we show how this operation is facilitated.

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

\[
\rho(S) = w_1|S.R_{in}| + w_2 weight(S.OR) + w_3 \sum_{r \in \text{R}, e}(\text{order(r)}).weight(r) + w_4 \log_2(|S.AS|)^{-1}
\]

Some of these factors can be merged to simplify the expression. The third term is already covering the first two, as it sums over all the weights of the rules. We introduce another scaling function \(c(.)\) that returns a weight for each rule based on its position. By incrementing all the \(c(.)\) coefficients, and specifically increasing the value of \(c(1)\), we can take care of the first and second terms respectively. The new adjusted \(c_{new}(.)\) will substitute the use of \(c(.)\) and eliminate the extra terms. Also, the testing density in larger segments will be naturally lower. Thus, it can be removed while its effect is still evident. The resulting expression would be:

\[
\rho(S) = \sum_{r \in S.R_{in}} c_{new}(\text{order(r)}).weight(r)
\]

E. Rule Weight

A segment’s weight is directly affected by the weights of contributing rules as shown above. The factors affecting the weight of a rule include (but not limited to) the following:

1) Rule area: The more specific the rule, the more critical it is and easier to test exhaustively.
2) Depth in policy dependency graph: A rule can be moved up/down the policy as long as no intersecting rule is encountered. For any rule, the number of related and preceding rules is an important factor in its weight.
3) Rule Features: A features is any property that can be identified by firewall implementors as a special case or a specific parameter in the filtering algorithm. For example, using TTL as a criteria is a feature, as well as source port. Also, a specific port value can be another feature if it is handled differently than the norm (e.g., some firewalls consider accepting port 443 as an implicit request to accept 80). A weight is assigned for each feature, and this weight is distributed uniformly over all rules containing it.

\[
\text{Weight}(R_i) = \alpha \parallel R_i.AS\parallel + \beta |\{r_j : (j < i) \land (R_i \cap r_j \neq \Phi)\}| + \sum_{f_i \in \text{Features}} w(f_i) |\{r_j : f_i \in r_j\}|
\]

| \text{Algorithm 2 ExtractPS (SEGLIST, PS}_\text{max}, \text{IterMax, SegCoverage)} |
VI. IMPLEMENTATION DETAILS

c) Segment and Space Representation:: The intermediate representation used for compiled rules as well as for segments was Boolean expressions over Ordered Binary Decision Diagrams (OBDD) [6], [13]. Our framework was developed using the C/C++ programming language and BuDDy, an OBDD package implemented in C/C++ [15]. It facilitates set operations (e.g., intersection, difference, etc), sample extraction or finding satisfying assignment, and space area calculation for segments and minterms.

d) Packet Handling:: The system has been implemented targeting Linux platforms. Several releases has been tested successfully, without compatibility issues. The packet creation was handled by, both, the libnet public library and direct building of packets for non-standard forms. Packet capture at the firewall output (i.e., the Spy module) was implemented via the libpcap library. The injection part had a parallel operational option to archive the created packets into a trace file for identical experiment regeneration, where the trace file can be loaded directly instead of going through the policy generation and packet selection process.

e) Spy Module:: The Spy is a stand alone program that resides behind the firewall. It sniffs outgoing packets, and collects them after receiving a test session ID in a “start test” request. All packets are tested for this ID, and if found they are marked in a bitmap. When the test is over, the Spy compares the expected with the actual firewall behavior and errors will be sent back to the engine for analysis. The bitmap that represents the expected behavior (sent from the Engine to the Spy), and the bitmap of the discrepancies (sent from the Spy to the Engine) are both compressed for transmission.

The Spy is built to handle several tests simultaneously, each can be coming from a different Engine. Thus, it has to be designed to support 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 and queues all packets with the system signature, (2) DeMux process: demultiplexes the packets 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.

f) Reporting Capabilities:: Reporting facilities has been placed in a separate module to enable independent analysis as well as encapsulation into other testing frameworks to be orthogonal to current system internal design. Two parameters are provided to control the report depth; Analysis and Verbose levels. The levels are treated incrementally (e.g., a verbose level of 3 includes all the features for levels 1 and 2), but not all level pairs (i.e., <verbose,analysis>) are valid. Some of the typical sections of a moderately verbose report: (1)Report and Scenario summary: Basic report statistics including total packets, different errors, with respect to the whole report and each scenario individually. Also, aggregates on the segment and rule levels are mentioned in scenario level summary. (2)Segments and Rules subsection: Statistics per each segment and rule in the scenario. Optionally the rules defining each segment are displayed as well as rule details. (3) Fields and Features subsections: Correlation figures between errors and the use of specific fields or features in rules. If the rules with a specific filtering field/feature tends to have matching errors, the strength of this tendency will be shown here.

VII. EVALUATION AND RESULTS

The system as a whole has been extensively used to test several images and releases of both IOS and PIX. Tests with policies as large as 30K rules, and test packets up to 256K packets have been used. Some of the pre-release/development OS images were selected especially for having known problems. Of course, the images were obtained under a strict NDA that prevents releasing what types of bugs have been reported and in what images.

The evaluation, with respect to 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.

A. Policy Generation Evaluation

The main goal in testing is to cover all possible configuration parameters that can be deployed on the test device. For thorough evaluation, we generated 200 different policies with policy sizes that range from 100 up to 24,000 rules. In this section, we evaluate the policy generator module by measuring the coverage of generated policies for possible configuration cases. Second, we will also measure the adherence of our policy generation in following the required guidelines and properties as provided by administrators such as policy size, rule complexity, field values, rule structures and the use of specific clauses. Third, we also evaluate the complexity/scalability of our approach.

1) Field coverage: The BNF-graph is a complete mapping of rule fields. We need to guarantee that each field has contributed to at least one rule during the testing process. This is a straightforward probability expression, where we can estimate the probability of having a single edge (or more) not selected in our graph. After generating more than 200 policies, the results showed that for non-trivial policy sizes (e.g., longer than 50 rules for complex grammars as Cisco IOS’s extended ACL format) all fields has been used (e.g., ports, TCP fields, ICMP qualifiers, etc).

2) Space coverage: Another important aspect is the coverage of values of individual fields. In other words, this means how diverse are the values selected as a criteria in policy rules for a specific field. However, some fields are binary in nature (e.g., TCP flags) and this metric is reduced to the previous one of field coverage.

By analyzing over 200 of our generated policies, we show that coverage for fields with non-trivial ranges is achieved for all possible values using ranges (e.g., subnetting). In fields that contain inequalities or range operators the results show almost complete value coverage (i.e., ports). To be conservative, “any” field values are removed as their coverage...
Empirically, we found the ratio to be less than 10 for almost all our test cases. For the policies more than 250 rules in our study, they never surpassed this bound. The subset of policies used in Fig 8 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 makes the overlapping between rules to be highly probable, which explains why there are no generated policies in our experimentation with \( |\text{Segments}(P)| \approx |P| \) (i.e., high proportion of distinct rules). If complex grammars are to be used, the number of segments will drop further and the ratio will approach unity.

In general, the segmentation algorithm may yield exponential output size and running time, particularly if the Boolean functions represent general expressions rather than firewall rules. However, in the case of firewall rules, multiple ACL inherent restrictions are applied on the Boolean variables. For example, it is not possible to specify the value of the \( i^{th} \) bit of an IP address without fixing the \((i-1)\) previous bits. Also, the variables representing some fields are either all mentioned in the Boolean expression of the rule or all omitted. This applies for the protocol, ICMP and IGMP qualifier fields, and ToS/precedence fields.

### C. Packet Selection Evaluation

We compare our packet selection module against the random sampler testing mechanism. The operation of the random sampler is as follows: given available testing time; the number
of possible test packets are sent, spreading them 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 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. It is worth mentioning, that to be highly conservative in our estimations, we selected only a single packet per segment (i.e., as if every segment consists of a single primitive segment). However, in actual test runs against firewall OS images that has been tampered with to force errors (as well as development-stage images with known errors from a major firewall manufacturer), using primitive segments provided enough coverage to discover the erroneous areas in every single test run we executed. Having more than 15 different types of induced/known bugs over 6 different operating system flavors and versions gave us confidence that adding the primitive segment feature was enough to provide release grade tested product.

Fig. 9(a) and 9(b) show the absolute and 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, we use the correlation between the two vectors (the weight function, and the actual error probability). It can be seen that any non-negative correlation gives a gain over random sampling. Even with zero correlation, it still gives better results because our technique samples within each segment guaranteeing a better distribution and ensures that no segment will be skipped. In contrast, random sampling might skip complete segments. Take into consideration that in these two graphs as well as in the next experiment, we were as conservative as we can; all tiny segments (and those with very high weights) were removed to smoothen the effect of heavy sampled segments. This would have caused the results to be more inclined to our technique.

This evaluation does not take into consideration the nature of common implementation errors; they mostly cause errors in whole regions rather than disperse them randomly in a segment or more. In other words, if there exist an error it is highly probable that whole segments (or rules) will be mishandled. This is supported by our observation of real problems. Thus, a single packet per segment will be enough to capture the error, while random sampling might not pick a sample from the affected areas. Simple calculations show the random sampler will miss hitting an erroneous segment with a high probability (∼ 100%) using millions of packets (Table 10). This probability is: \( P(\text{miss}) = (1 - 2^{-s-S})^N \), 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. Our technique perform quite well, in all policy styles. 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 random sampling counterparts. Thirdly, to include the effect of small segments, the fourth graph shows how 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 better results over random sampling when all the features in the tested firewall/policy are used (e.g., exhaustive testing of small segments, high sampling for high weight/moderate weight segments, etc). Moreover, adding primitive segments to the packet selection criteria gave even better results in our test environment.

VIII. CONCLUSION AND FUTURE WORK

In this paper, we present an automated framework for validating correct implementation of ACL configuration in firewalls. Our framework is fully automated and applicable to many ACL standards with minimal changes. It provides random policy generation, traffic generation and injection, traffic capturing and analysis, and test result reporting. Our policy and traffic generation obtain high coverage of configuration profiles and test space respectively, without using exhaustive testing. Random policies are generated based on custom profiles or learned properties from given manually-written policies over the grammar of a specific ACL. Our policy generator considers all important policy configuration properties such as rules complexity, distribution of fields values, rule interaction, policy size, fields dependency and others. The test traffic generation technique avoids the exhaustive testing via efficient representative packets based on policy coarse and fine-grain segmentation that results in complete coverage of all decision paths of a filtering device.

The evaluation results show that it is not necessary to know beforehand the error model of the implementation to be able to efficiently discover the existence of implementation problems. 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. We also show high coverage of field combination and values which is necessary for achieving low false negative.

Currently, research is in progress to enhance the policy generator to incorporate more options and capabilities to generate human like policies, and to target specific operational problems. Moreover, generating independent and orthogonal policies (with respect to the filtering algorithm) renders consecutive policy generation an increasingly hard problem. Studying the segmentation behavior for several policy styles needs further investigation. Also, tweaking the system parameters for different filtering techniques can enhance the framework performance and overall testing time.

ACKNOWLEDGEMENT

Special thanks to Khalid Elbadawi and Bin Zhang for their efforts in evaluating the system.

REFERENCES

Ehab Al-Shaer is an Associate Professor and the Director of the Assurable Networking Research Center (ARC) in the School of Computing at DePaul University. He received his Ph.D. and M.Sc. in Computer Science, and B.Sc. in Computer Engineering from Old Dominion University (USA), Northeastern University (USA) and KFUPM (KSA) in 1998, 1994, and 1990 respectively. His primary research areas are network security, distributed monitoring, fault diagnosis, and multimedia transport protocols. Prof. Al-Shaer edited/co-edited more than 10 books and book chapters, and published more than 80 refereed journals and conferences papers in his area. Prof. Al-Shaer is the General Chair of ACM Computer and Communication 2009-2010 and NSF Workshop in Assurable and Usable Security Configuration, August 2008. He served as a member in the technical program and organization committees for many IEEE and ACM conferences.

Adel El-Atawy is a PhD candidate in the School of Computing at DePaul University. He received his M.Sc. and B.Sc. degrees in Computer Science from Alexandria University, Egypt in 2003 and 2000 respectively. While in Egypt he also co-founded eSpace, a leading software house in the Middle East. Currently, he is a research assistant at the Assurable Networking Research Center (ARC) at DePaul University, and a lecturer on leave from Alexandria University. His research revolves around using information theoretic analysis and statistical techniques in network security systems.

Taghrid Samak is a PhD student in the School of Computing, at DePaul University. Her research is focused on policy modeling, analysis and generation. She finished her Bachelor degree in Computer Science from Alexandria University, Egypt in 2004 where she has been awarded a graduate fellowship for academic excellence. She joined the Assurable Networking Research Center (ARC) at DePaul University in Fall 2005. Taghrid is an active member of DePaul chapter of Upsilon Pi Epsilon. She received the 2008 Dan Drew award from National UPE for her services.