Towards Network Security Policy Generation for Configuration Analysis and Testing

Taghrid Samak
School of Computing
DePaul University
Chicago, IL
taghrid@cs.depaul.edu

Adel El-Atawy
School of Computing
DePaul University
Chicago, IL
aelatawy@cs.depaul.edu

Ehab Al-Shaer
School of Computing
DePaul University
Chicago, IL
ehab@cs.depaul.edu

ABSTRACT
Access-control lists are an essential part in the security framework of any system. Researchers are always in need to have a repository of ready made policies for conducting research and development. Such policies, especially firewall policies which are the focus of our work, are needed to perform performance testing as well as configuration analysis.

In this paper we introduce a novel technique to perform access-control policy generation. The proposed approach learns policy parameters from a set of given policies. It generates policies that conform with natural policy-writing practices while following the grammar syntax required by the security device. A probabilistic learning approach is used to infer transition probabilities for the policy grammar.

Categories and Subject Descriptors
C.2.0 [COMPUTER]: COMMUNICATION NETWORKS—Security, protection

Keywords
security policy, policy generation, policy grammar

1. INTRODUCTION
Access control policies are an essential part of many services and network devices in almost every aspect of our current increasingly complex networks. Many research areas that make use and revolve around policies are in need for a rich supply of policies and access control lists. Research in areas related to their functionality, verification and deployment suffer the lack of such repositories. In general corporate, network administrators consider their networks’ policies as an extremely private piece of information. Even after address anonymization, the structure of the policy might reveal some information about the network topology and layout, as well as the personal style of the administrator. Therefore, a means to generate these policies for research and development purposes is crucial for those research areas.

In order to be able to generate policies that are not just a random assignment of filtering criteria, we need to have some sort of a template or a study of actual policies structures over which we can build our synthetic policies. However, a question might arise that generating policies that imitate some given policies might not add much to our policy repository. The answer is two-fold. First, the generated policies can combine the properties of more than one policy, thus causing the generated policies to have a feel of human-written policies without imitating a specific policy per se. Second, by collecting the aggregate information about the writing/use patterns of specific criteria fields and terms we will have how a natural policy looks like. Afterwards, we can tweak these parameters as needed to look anywhere from a completely random structure to a policy that perfectly conforms with common practices.

Access control policies play an important rule in almost all network devices and services. They consist of an ordered list of criteria and actions. An administrator will set these criterion-action pairs to match the traffic to be accepted or dropped. For example, firewalls are placed at network boundaries to filter out unwanted incoming connections and traffic depending on the network’s running services/hosts and the expected users. Also, similar policies are deployed in host-based firewall applications and almost all modern routers have filtering capabilities as an optional feature to activate on individual ports. Different application areas stem from policy deployment on security devices, where research is conducted to perform optimization, testing as well as configuration analysis. For this wide area of applications, test policies are needed for the evaluation of any new development in policy algorithms. We present here two major application areas for our generation, and then discuss the generation requirements for such applications.

Policy configuration analysis. Tools designed for policy analysis are very crucial to administrators where human errors and misconfigurations need to be detected and resolved. Random policies cannot provide control on test cases that specifically require misconfigured policies. For example, many algorithms have been developed addressing the problem of identifying policy anomalies in firewalls as well as IPsec. ([1, 9, 22]). They have targeted misconfiguration resulting from rule interactions such as shadowing, generalization and exceptions. Other general analysis tools focus on property analysis and queries [7, 11, 13, 21]. To be able to
assess the effectiveness of such tools, a comprehensive set of filters or policies need to be generated that are relevant to the domain under test.

**Security devices testing.** The second application that needs comprehensive policy evaluation is the testing of security devices. Implementation correctness is the main concern when building new security hardware, or installing new system implementation. Apart from testing the filtering performance, the actual implementation on the network device needs to be tested. Here is where generating wide range of policies with different features comes in place. Theoretical [12, 21] as well as practical [2, 8, 19] testing approaches involve having a set of policies that can be modeled or applied to the security device. For both approaches, test cases can be generated either to target specific flows or to cover all possible device configurations. For the purpose of general testing not for certain vulnerabilities, test case generation is needed that reflects network specification as well as guarantees domain coverage.

1.1 Generation requirements

Studying various application domains the following issues need to be addressed in the generation process:

- Policy features in terms of filtering information and packet fields should be satisfying application requirements (e.g. protocol-port field dependence).
- Domain information needs to be implicitly embedded into the generated policies without extra processing for different domains.
- Policies with different syntax and structure need to be generated and customized for different vendors, in order to be deployed easily for testing.

1.2 Our contribution

We address in this work the previous generation requirements, where policies will be generated based on a specific grammar supported by the targeted application or security device. Domain information will be learned as policy grammar transition probabilities. From a given set of policies the grammar/syntax probabilities will be generalized to produce a larger set of policies following the same grammar and probability distributions. According to a given syntax, policy features will be learnt from provided samples. The probabilities of each transition in the policy grammar can be learned from a set of previously defined policies, or passed to the system for direct generation. This approach will specifically cover the common practices by administrators, and the domain information.

Using an initial set of policies from the domain on interest will help learning domain information as well as policy feature behavior. It is also important in testing different configurations following the same domain behavior. Another important advantage in our generation methodology is that it provide domain transparency. The developed tool can be used by administrators in each domain without having to give away their policies. The tool will learn the domain specific features in the privacy of the domain, not at the development phase.

The proposed learning and generation approach was applied to learn features of a set of IOS policies provided by cisco. We have also developed a general framework to facilitate learning different grammars (e.g. IPTables).

**Paper layout.** The following section (2) will present an overview of policy grammar/syntax model. Theoretical foundation for the learning process will be provided in section 2. In section 3 the generation process will be discussed in detail with different generation options. The implementation of the proposed approach is presented in section 4, with its evaluation results in section 5. A detailed discussion of related work in policy generation for different applications and how they compare to our generation approach will be presented in section 6.

2. POLICY GRAMMAR MODEL

In general a policy or a filter set is an ordered list of rules, $R_1$, $R_2$, $R_n$. Each rule is defined as $R_i : C_i \rightarrow A_i$ , where $C_i$ is the rule condition that filters incoming traffic, and $A_i$ is an action applied to traffic satisfying the condition. In most cases, $C_i$ is expressed in terms of packet header fields, typically source/destination IP addresses, source/destination port number and protocol. Sometimes session information is included for devices with layer 4 capabilities. We will restrict our analysis here to the aforementioned five fields (total of 104 bits in the packet header).

The rule-condition pair is translated into specific syntax to be deployed at policy enforcement points (devices). This constitutes the actual configuration representing the logical security requirements. The exact policy configuration depends on the operating system version on the security device. For example, cisco routers or firewall might use IOS or PIX for defining filtering or routing policies. Most Unix/Linux boxes may use NetFilters and IPTables configurations. Each of those policy definition languages has its own commands and syntax. We are interested here in covering the syntax for rule set definition. For this purpose, rule-action pair will be translated into sequence of strings and values representing different fields in the packet header as well as an action.

The action part of the rule depends on the application domain. A general access control list will have $A_i \in \{accept, deny\}$. This is also the first step in classification or routing where traffic has to be accepted first then extended actions can be applied.

We regard rule syntax to be matching a context free grammar (CFG) that accepts every rule string. A learning process is developed to assign probability distributions to the policy grammar resulting in a probabilistic CFG (PCFG [3, 14]).

2.1 Probabilistic Context-free Grammar

We will first consider a simple policy with only the basic five packet header fields used in the filtering (source/destination IP, source/destination port, and protocol). Following the
Let the policy context free grammar be defined by the model \( M = (N, T, R, S) \), where \( N \) is the set of non-terminal symbols ("access – list", integers, \ldots), \( T \) is the set of terminal symbols ("permit", integers, \ldots), \( R \) is the set of production rules (lines in the grammar definition), and \( S \in N \) is the start symbol. Assuming the sets \( N \), \( T \), and \( R \) to be finite, the number of parse trees resulting from the grammar is also finite (which is the case for policy rules). The goal of the generation process is to first estimate the probability of each production rule, and then use those probabilities to generate multiple policies for specified features. 

Let \( \Omega \) be the set of all possible finite parse trees generated from a grammar \( G \). Those are all possible combinations of rules that are accepted by the policy syntax. For each subtree \( \tau \in \Omega \), the number of occurrences of a production rule \( (A \rightarrow \alpha) \) in \( \tau \) is denoted by \( f(A \rightarrow \alpha; \tau) \), and \( f(A; \tau) \) is the number of occurrences of \( A \) in \( \tau \). For example, the rule access-list 50 permit tcp any any eq 80 has the following production rules in its parse tree: \( N \rightarrow 50 \), \( A \rightarrow \text{"permit"}, P \rightarrow \text{"tcp"}, IP \rightarrow \text{"any"}, Op1 \rightarrow \text{"eq"}, Y \rightarrow 1 \ldots 65535.

Policy generation should follow the probability distribution of common policy configuration. In our generation, two modes of operation decide the probability distribution of the production rule for the PCFG. The static random mode is based on well-known policy properties that can be used to set the probabilities for the rules. The second mode of operation is derived from a given repository of policies, and our algorithm learns the probability distribution for the PCFG. Relative Weighted Frequency method is used in the learning process.

### 2.2 Relative Weighted Frequency

Given a finite set of samples (policy rules), we wish to estimate the production probabilities for the policy grammar. Each rule corresponds to a parse tree \( \tau \). If the domain is fully observable (all possible rule combinations can appear in the policy), then the maximum likelihood estimation would be sufficient to approximate the production probability. The probability of rule \( (A \rightarrow \alpha) \) given \( n \) parse trees \( \tau_1, \tau_2 \ldots \tau_n \) (\( n \) policy rules) can be estimated as:

\[
\hat{p}(A \rightarrow \alpha) = \frac{\sum_{i=1}^{n} f(A \rightarrow \alpha; \tau_i)}{\sum_{i=1}^{n} f(A; \tau_i)}
\]

(1)

The relative frequency estimation is used in the case of incomplete observations where the expectation maximization algorithm is used [3]. This is more appropriate for the case of given policy rules since not all possible combinations can appear in a single policy within one domain. The procedure for estimating production probabilities for partially observable languages follows from [3]. A subset \( \omega \subseteq \Omega \) is a finite set with every production rule appearing in the trees in \( \omega \). A positive weight \( W(\tau) \) is assigned to each tree \( \tau \in \omega \) such that \( \sum_{\tau \in \omega} W(\tau) = 1 \). The system production probabilities are then defined by:

\[
p(A \rightarrow \alpha) = \frac{\sum_{\tau \in \omega} f(A \rightarrow \alpha; \tau) W(\tau)}{\sum_{\tau \in \omega} f(A; \tau) W(\tau)}
\]

(2)

For our generation purpose, the set \( \omega \) is the given set of policy rules. \( W(\tau) \) is calculated by the probability that a specific parse tree \( \tau \) appears in \( \omega \) (probability of field values belonging to the rule yielding \( \tau \)).

### 3. THE GENERATION PROCESS

The input for the generation process includes policy syntax description (grammar), and a set of policies following the same grammar. A learning process is then performed to capture policy properties from the given policy set taking into consideration the provided syntax.

The generation approach first constructs the PDA (Push-Down Automaton) for the given grammar, and the learned probability distribution is used to guide the edges of the PDA in the generation. The PDA is defined as directed graph with the overall expansion of the production rules of 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 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. The main difference between the two modes is the means by which production probabilities are assigned to transition edges. The
ultimate goal in assigning graph probabilities is to follow equation 2, and satisfy the following relation for each rule string \( d = \pi_1\pi_2\ldots\pi_m \) corresponding to parse tree \( \tau \):
\[
W(\tau) = \Pi_{i=1}^{m} p(\pi_i)
\] (3)

3.1 Static-Random Mode
This modes is the default for the system, where some general information is provided to the generator. The probabilities on each link are provided to the system before-hand in the grammar. Those probabilities are used in the case of multiple branches (\( \{ \) ). For generating a policy, more auxiliary parameters are needed as well: 1) policy size: the average number of rules that will be generated for each policy, 2) average rule complexity: this parameter is the probability for an optional field to be used. The higher the value the more complex the rule will be. Another set of parameters are used to control the way the generator selects values for different fields. 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 desirable and can be optionally triggered.

3.2 Informed Mode
This generation mode utilizes the relative weighted frequency estimation approach in learning the transition probabilities on the graph edges. In this sense, it 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. In this way, the generation is more likely to produce human like 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 policies provided by cisco, with number of rules varying from 50 to 10000.

Level-I. For each given policy, the graph is traversed as if the policy is being parsed for correctness. A counter is maintained at each link of the graph, that will be incremented for every 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 will reflect fields interactions, in contrast to prior work that estimated single field probabilities [18].

Level-II. An important observation about policies used in our study, is that the position of the rule within the policy affects rule structure. Specific fields are defined in rules at the beginning of the policy, while general values and wild cards are more likely present at the end. To be able to learn this property, probabilities for different parts of the policy are learned independently. Each policy is partitioned to a number of rule groups (consecutive rules). Probabilities for each partition are learnt separately, following the same procedure from level-I. If two consecutive groups have similar distribution they may be merged. The question now is where to put the partition boundary (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 \( G'_c \gg G_c \gg T \) 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. Algorithm 1 describes the process.

The \( \text{UpdateStat}() \) updates the statistics on the graph based on the current rule. The discrepancy measure used is the euclidian distance between the weights of the edges in both graphs (since they have the same structure). The final constructed graph is depicted in figure 2. If the policy resulted in \( m \) partitions, then there will be \( m \) graphs with different probability distributions defined for the policy. The portion of the policy that will be generated for each partition is proportionate to the learned partition sizes. The weights are estimated by the relative number of rules contributing to the subgraph \( (r_c) \) normalized over the total number of rules.
\[
p(G_c) = \frac{r_c}{\sum_{j=1}^{m} r_j}
\] (4)

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.

4. IMPLEMENTATION DETAILS
Here, we first summarize the implementation details for the syntactic policy generation, and then apply the relative weighted frequency method to learn production probabilities for the policy grammar. The first step is understanding the required policy syntax. The grammar is parsed into a push down au-

Algorithm 1 Informed-Level-II (R: set of rules, T: threshold)
91: \( G \leftarrow \lambda \) \{Initialize graph list\}
92: for all \( r \in R \) do
93: \( G'_c \leftarrow \text{UpdateStat}(r, G_c) \)
94: if \( G'_c \gg G_c \gg T \) then
95: \( \text{addGraph}(G, G_c) \)
96: new \( G_c \)
97: end if
98: \( G_c \leftarrow \text{UpdateStat}(r, G_c) \)
99: end for
Figure 3: A truncated version for the IOS extended access-control list syntax.

4.1 Policy Grammar

A general grammar is needed that can both define syntax and help translation to domain information. Embedded annotations are used to describe these semantics to the grammar parser and the policy generator afterwards. For example, without added annotations there will be no distinction between source and destination IP field in the final expansion of the grammar. Also, keywords in the syntax should be mapped to either a value (e.g., “tcp”=6, “domain”=53, etc), or a specific operation (e.g., “eq”, “gt” etc.). Moreover, cross-field constraints need to be defined (e.g., specific port-protocol dependencies).

omitted. Also, TCP flags rules are removed for clarity. Some dependencies exist between the use of L4 tokens and the “fragment” keyword are not included. For a full version, please refer to the authors project page http://www.arc.cs.depaul.edu/~taghrid/PolicyGen. In our implementation, annotations are prefixed with a backslash. Special keywords/operations in our grammar language are:

- \n FieldID(n): A predefined number that identifies the physical packet field.
- \{num(n1,n2): An integer range from n1 to n2. Acts as a shortcut instead of listing all the valid numbers as would be the case in standard grammar.
- \V(x): The value of the current token is x. For example, “tcp”\V(6) means “tcp” has the value of 6.
- IPvalue and IPmask: Special handling for IP, and mask.
- \Trans: Provides extra translation for some tokens. For example, conditions with inequalities over port

4.2 BNF Graph

The BNF graph is a directed graph representing the FSA of the grammar. The graph has a single starting state, and a single final accepting state. Our generation technique uses it to generate rules that are guaranteed to follow the required syntax.

Each non-terminal in the grammar is expanded into a subgraph between two nodes (including the initial non-terminal \s that builds its associated graph between the sole starting and ending states of the graph). Graph edges store information about field IDs, dependency between fields, probability of each edge, and other annotations provided in the grammar. Each edge contains the necessary information to decide a part of the syntax: <s,v,f,c,p>. s is the string text to be matched, v is the value to be assigned if matched and f is is the packet field. The fourth value, c, specifies any cross-field

9Extra definitions for ICMP and IGMP types/codes are omitted. Also, TCP flags rules are removed for clarity. Some dependencies exist between the use of L4 tokens and the “fragment” keyword are not included. For a full version, please refer to the authors project page http://www.arc.cs.depaul.edu/~taghrid/PolicyGen.
dependencies (which is itself a list of \(<\text{field-value}>\) pairs). The last item is the probability of this outgoing edge.

4.3 Collecting Field Value Distribution

In the process of studying the distribution of rules, a simple frequency counter per edge can be generally enough. One can add a few tweaks such as using LaPlace estimator \(\hat{p}(G) = (r+1)/(m+\sum_{j=1}^{m} r_j)\) instead of normal probability estimation that will help avoid having zero probabilities in the final model.

In order to calculate the distribution of the values themselves, we need to keep track of the specific value chosen in each edge. Note that some edges might result in a different value each time it is traversed (\(e.g\.), edges based on lookup tables, random numbers, etc). Once an edge is traversed that results in a value (\(e.g\.), protocol, port, etc), a specific container will be updated. Each container is responsible for a specific field. The FieldID() annotation will be used to identify which container is to be updated. If a container is set for a grammar rule, all values that results from traversing the associated subgraph will update the container accordingly.

A container is basically a hash table. Once a value is found, the container number as mentioned in the FieldID() will be fetched. If the value does not exist in the hash table, it will be added, otherwise its counter will be incremented.

An extra annotation \(\backslash Pbin(container, value)\) can be used to add an additional container number to be updated with the value given. This is used to keep track of features that does not translate to a value \textit{per se}. For example, such annotations can be used to see how many a port is mentioned by name from a lookup file versus by number. Another example can be keeping track of how many times the matching operator with the port number was “eq”, “range”, “gt”, etc. Also, it can be used to keep track of other relations that does not appear in the grammar. For example, number of times a specific IP host was mentioned in a rule, where the protocol was set as “any”. These relations can help develop more complex studies about inter-field dependencies.

These additions facilitate further analysis and studies to existing policies without revisiting the analysis code; editing the grammar is all that is needed.

5. POLICY GENERATION RESULTS

As described in section 3.2, the operations of level-I and level-II are performed, level-I learns the probability over all graph rules to generate a single graph. We will use here a set of policies provided by cisco, ranging from 50 – 10000 rules. Those policies are written in the standard IOS format (3). The syntax file for the given policies is first provided, which constitutes the complete version of the rules depicted in table 1.

5.1 Level I

We will present here the learned probability distributions for the policy grammar. Each production rule will result in a certain parse tree with each nonterminal expansion contributing to the overall probability of the production rule.

Following the given syntax, the first two production rules to be analyzed are \(\text{Action} \rightarrow \text{"permit"} \mid \text{"deny"}\) and \(\text{Proto} \rightarrow \text{Byte} \mid \text{udp} \mid \text{tcp} \ldots\). According to the provided policies the protocol is always belonging to the set of known names, particularly (tcp, udp, icmp and ip). Other protocols never showed in the given policies. The probability distributions of these production rules are summarized in tables 2 and 1.

![Table 1: Action Distribution](image)

<table>
<thead>
<tr>
<th>Action → &quot;permit&quot;</th>
<th>p(A → α)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Action → &quot;deny&quot;</td>
<td>0.03</td>
</tr>
</tbody>
</table>

![Table 2: Protocol Distribution](image)

<table>
<thead>
<tr>
<th>Proto → tcp</th>
<th>p(A → α)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Proto → udp</td>
<td>0.25</td>
</tr>
<tr>
<td>Proto → ip</td>
<td>0.03</td>
</tr>
<tr>
<td>Proto → icmp</td>
<td>0.03</td>
</tr>
</tbody>
</table>

The following set of rules in the policy grammar defines the IP and port syntax. For IP addresses, the probability estimation involves more processing since general IP rules appear twice as source or destination. The same applies also for port numbers. Table 3 summarizes the calculations using the relative weights as defined in equation 2. The different instances of IP or port definitions that resulted in different parse trees will have different probabilities. Each production rule probability will be approximated as shown in the table, and the generation will use those probabilities when traversing the graph.

For the given set of policies, the probabilities of having specific hosts (\(I\text{P}host \rightarrow \text{"host"} \mid \text{IP}\)) or a general IP address (\(I\text{Pany} \rightarrow \text{"any"}\)) as source or destination IP is almost the same. For port values, equality and ranges dominate the policy rules. This shows that the learned probabilities will follow the given policy structure and field distribution. Other test cases will depend on the domain values provided.

It is important to note that those leaned probabilities will be applied at the single transition from graph nodes. Parsing a complete rule will result in multiplying all edge probabilities to find the rule weight \(W(τ)\).

For a given set of policies, the probability of generating a specific policy rule can now be estimated. For example, the probability of generating the following rule can be estimated

![Table 3: Production rules probabilities](image)

<table>
<thead>
<tr>
<th>X → α</th>
<th>p(X → α)</th>
</tr>
</thead>
<tbody>
<tr>
<td>I\text{P}host → &quot;host&quot; \text{IP}</td>
<td>0.48</td>
</tr>
<tr>
<td>I\text{P}any → &quot;any&quot;</td>
<td>0.37</td>
</tr>
<tr>
<td>I\text{P}pair → \text{IP} IP</td>
<td>0.14</td>
</tr>
<tr>
<td>Port → &quot;eq&quot; \text{Y}</td>
<td>0.76</td>
</tr>
<tr>
<td>Port → &quot;range&quot; \text{Y} \text{Y}</td>
<td>0.18</td>
</tr>
<tr>
<td>Port → other</td>
<td>0.06</td>
</tr>
<tr>
<td>Byte &quot;tcp&quot; &quot;host&quot; \text{IP}</td>
<td>0.69 \times 0.48</td>
</tr>
</tbody>
</table>
Our focus in this work is to address the problem of generating the proper rule set for different tests used in firewall benchmarking, or general security device testing.

Feldmann and Muthukrishnan, [5], identified some requirements for testing packet classification algorithms relating to rule set definition practice. Those specifications or metrics included policy size, rule dimensionality, rule structure, and policy updates. The rule dimensionality specifies how many fields are used to define the rule. To evaluate their approach, [5] used data sets derived from traffic collected on edge routers using Cisco NetFlow statistics. In NetFlow, a source-destination IP pair define a specific flow within a certain time interval. The generated statistics also include masking information, which is useful in writing policy rules corresponding to the observed flow. Three main features defined data set structure, and were used to compare different derived policies; the number of unique source-destination network pair (using masking information), the number of unique source IP and the number of unique destination IP. An artificial data set was also generated using routing information, and building rules from random combinations of different source-destination network pairs.

Some techniques have been developed to derive a firewall policy from either log files [6], or by actively probing the firewall with tailored packets [16]. None of these approaches related to our generation. Their main goal is to discover the deployed policy, while our aim here is to generate multiple policies with certain features.

In [20], Woo has proposed some characteristics for test data set for filtering and classification. The characterization depended on the location of the filter set and purpose of the machine. The size of the policies used varied from 8K to 128K. Each policy might have different partitions according to the machine location and purpose. The evaluation of the filtering approach proposed in [20] assumed using an ISP edge router. Filters were randomly generated for VPN, and ingress filters with different specifications (single subnet, multiple subnets, and entire domain).

For the purpose of generating policies that are similar to a previous rule set, ClassBench project [18] is one of the most popular framework that cover this. It has been widely adopted by several applications since its launch in 2005 ([15, 17, 23]). ClassBench relies on a previously defined filter set (seed) to approximate field statistics. Three filter set types were studied (access control lists, firewalls, and IP chains). The derived statistics are then used to generate more general policies and traffic to test those policies. The parameters used in their configuration file provide high degree of flexibility and tunability. Another similar approach was presented in [4], where random policies are generated, and then an efficient mechanism is used to generated traffic from a policy. Although both frameworks [4, 18] support using probability values for packet header fields and individual rule structure to generate policies, they did not consider interaction between multiple fields for each rule. This is implicitly covered by our approach, where the learning process inherit the dependency between field probability distributions.

### Table 4: Rule probability example

| G1   | permit  icmp  icmp-types | 7%   |
| G2   | permit  tcp  any  any  eq  specific-ports  | 20%  |
| G3   | deny    ip    all  specific-ports          | 3%   |
| G4   | permit  specific rules                  | 40%  |
| G5   | permit  all  specific-ports             | 15%  |

Table 5: Level II partitions

#### 5.2 Level II

For level-II analysis, where the policy is partitioned into different graphs we also used the same set of policies. The given set of policies is divided into two groups: small and large policies. Small policy sizes are those having < 100 rules, while large policies may reach 10000 rules. For the small policy group, the partition structure is summarized in table 5. The large group has similar structure but with different rule ratios, where the specific rules occupy more than 50% of the policy.

#### 6. RELATED WORK

As presented in section 1, various applications domains needs and have used synthetic policies to evaluate their approaches. Only a few tried to come up with a general process that provide policies with specific features. Before going into details about how policy generation has been addressed in the context of specific domains we will review some standards proposed in the IETF. In RFC 3511, [10] and RFC 2647, benchmarking firewall performance was addressed. The two proposals focused on testing a security device including all performance aspects; load, latency, configuration ...etc. One of the elements considered for setting up firewall performance tests is the Rule Sets; the collection of access control polices that determine which packets are accepted or denied. In [10], it was recommended for the test to include a rule for each possible hosts or virtual clients. Different size rules sets (policies) are also recommended for comparison. The proposal also addressed briefly the test traffic in relation with the rule set. The traffic should match rules configured at the end of the rule set and not the beginning. This is also equivalent to having any extra rules at the beginning of the rule set, maybe to assess the performance of the device with worst configuration. Finally, the RFC 3511 required to have a rule that denies all traffic not matching any defined rule in the policy. The rest of the proposed benchmarking specification focused on device throughput sometimes specifically to TCP sessions. Tests are also proposed for latency and resiliency to attacks.
7. CONCLUSION

Synthetic policy generation is a very important tool in the analysis and verification of security device configurations as well as testing device’s continuously changing implementation. This work addresses the generation of security policies that follow a given grammar/syntax and learning grammar transition probabilities for generalized generation of policies. A policy is modeled as a set of rule strings that follows a probabilistic context-free grammar. Our learning methodology updates grammar probabilities to provide more policies in the generation process. This process is split into two levels: level I focuses on learning rule probabilities reflected in field values of packet header data. Dependencies between those fields will be captured in our learning methodology. In level II, an overall structure of the policy is learned with different policy partitions corresponding to different probability distributions. Both levels of operations will reflect domain information as well as administrators’ practice. The learning and generation approach was used on real policies provided by a major network provider, and their resulting probabilities were presented.

The proposed implementation also provides flexibility in defining various policy grammars and device dependent languages, which makes it ready for use in different domain without extra development efforts.

Future extensions to this work include:

- Refining the learning process to have a better recognition of different phases of the policy as well as linking the learnt patterns to the domain information and network topology.
- Using different policy sets as seeds from other domains for the generation and evaluating their distribution with respect to domain information. This could be performed at the target domain to avoid any privacy issues from sharing the initial policies.
- Investigating the process of learning the syntax without having the policy grammar. The learning approach needs to be adjusted for learning the overall grammar with only policy files.

8. REFERENCES


