Misconfiguration Analysis of Network Access Control Policies

by

Tung Tran

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2008

AUTHOR'S DECLARATION

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Network access control (NAC) systems have a very important role in network security. However, NAC policy configuration is an extremely complicated and error-prone task due to the semantic complexity of NAC policies and the large number of rules that could exist. This significantly increases the possibility of policy misconfigurations and network vulnerabilities. NAC policy misconfigurations jeopardize network security and can result in a severe consequence such as reachability and denial of service problems. In this thesis, we choose to study and analyze the NAC policy configuration of two significant network security devices, namely, firewall and IDS/IPS.

In the first part of the thesis, a visualization technique is proposed to visualize firewall rules and policies to efficiently enhance the understanding and inspection of firewall configuration. This is implemented in a tool called PolicyVis. Our tool helps the user to answer general questions such as ''Does this policy satisfy my connection/security requirements''. If not, the user can detect all misconfigurations in the firewall policy.

In the second part of the thesis, we study various policy misconfigurations of Snort, a very popular IDS/IPS. We focus on the misconfigurations of the *flowbits* option which is one of the most important features to offers a stateful signature-based NIDS. We particularly concentrate on a class of *flowbits* misconfiguration that makes Snort susceptible to false negatives. We propose a method to detect the *flowbits* misconfiguration, suggest practical solutions with controllable false positives to fix the misconfiguration and formally prove that the solutions are complete and sound.

Acknowledgements

First of all, I would like to thank my supervisor, Dr. Raouf Boutaba, for his support, guidance, and supervision. He has been very professional and understanding. His encouragement has motivated me tremendously to complete this thesis.

I am also deeply indebted to Dr. Ehab Al-Shaer, from DePaul University, for his co-supervision, technical guidance, and professional advices throughout my research. His assistance has helped me enormously to advance this research.

Special thanks to my friends who support me during my study at the University of Waterloo.

To my dear parents !

Table of Contents

List of Figures

List of Tables

Chapter 1
Introduction

1.1 Network Access Control Systems

At the core of an access-control system is the secure evaluation of whether an established identity has access to a particular computing *resource* [7]. Access control is decided over an existing security context and a controlled resource. Modern access-control mechanisms are based on the *reference monitor* concept introduced in early 1970s by Lampson [24]. A reference monitor is the trusted computing base component of a computing system that mediates every access of a subject to a resource in accordance with a security policy that governs such access. The policy normally has the form of rules and attributes associated with a registry of subjects and a registry of objects. The rules can be static access rights (permissions), roles, or dynamically deduced rights [7].

Network Access Control (NAC) is a computer networking solution that uses a set of protocols to define and implement security policies on all devices seeking to access network computing resources [30] . A NAC system controls access to a network with policies, that includes pre-admission endpoint security policy checks and post-admission controls over users' activities in a network. Even though the definition of NAC is evolving and controversial, the common goals of a NAC system generally include:

- *Mitigation of zero-day attacks:* a NAC system is able to reduce the damage of zero-day attacks by preventing end-stations that lack antivirus or patches from accessing the network and placing other computers at risk.
- *Policy enforcement:* a NAC system allows an admin to define policies, such as the types of computers or roles of users allowed to access the network, and to enforce them in network devices.
- *Identity and access management:* a NAC system enforces access policies in terms of IP addresses and/or authenticated user identities.

1.2 Network Access Control Policy Misconfigurations

Threats are made possible due to *vulnerabilities,* also referred to as *weaknesses,* either in the mechanisms enforcing a particular security policy or in the operational controls of that policy (such as those having to do with configuration parameters) [7]. For a network access control system to be

effective, the policies it supports must match those that its users need. Sometimes we need to access a resource that should be allowed, but that is denied by the system. This is typically the result of a NAC policy misconfiguration.

In [5], the authors defined two types of policies: *Implemented policy* and *Intended policy*. *Implemented policy* is the policy explicitly enacted via credentials that grant authority to users. *Intended policy* includes implemented policy and also policy that is consistent with user's intentions but has not yet been enacted through credentials. Policy misconfigurations are then defined as inconsistencies between *implemented* and *intended* policies.

To be more specific, in this thesis, we consider a NAC system policy misconfiguration as *any error or mistake in the policy that makes the system misbehave or perform poorly.* A NAC system misbehaves when it does not carry out the policy exactly as the system admin expected, and performs poorly when it takes longer time than it should to carry out the policy.

With the increased semantic complexity of NAC policies and the large number of rules that could exist, the possibility of policy misconfigurations has significantly increased. NAC policy misconfigurations can result in user frustration and wasted time, and can sometimes jeopardize network security. As a result, the process of identifying and fixing policy misconfigurations is essential to improving the usability and security of any NAC system.

NAC brings together a variety of network security systems including identity management, firewalls, IDS/IPS (intrusion detection system/ intrusion prevention system), anti-virus software, etc. However, in this thesis, we concentrate on studying policy configuration of firewalls and IDS/IPS because of their popularity and their importance in network security.

1.3 Introduction to Firewall

With the increase of network attack threats, firewalls are considered effective network barriers and have become important elements not only in enterprise networks but also in small-size and home networks. A firewall is a program or a hardware device designed to protect a network or a computer system by filtering out unwanted network traffic.

There are two main types of firewalls: network layer and application layer firewalls. The first type makes decisions on packets based on examining the TCP and IP headers. The second type works on the application level of the TCP/IP stack (i.e., all browser traffic, or all telnet or ftp traffic), and may intercept all packets traveling to or from an application.

Network layer firewalls generally fall into two sub-categories, stateful and stateless. Stateful firewalls maintain context about active sessions and use that "state information" as one of criteria to decide if a packet is allowed to continue its path. However, stateless firewalls do not keep track of connection states and provide network access control based on several pieces of information contained in a packet's TCP and IP headers.

In this thesis, we mainly concentrate on stateless network layer firewalls, also called stateless packet filters. The filtering decision is based on a set of ordered filtering rules written based on predefined security policy requirements.

A firewall security policy is a list of ordered filtering rules that define the actions performed on matching packets. A rule is composed of filtering fields (also called network fields) such as protocol type, source IP address, destination IP address, source port and destination port, and an action field. Each network field could be a single value or range of values. Filtering actions are either to *accept*, which passes the packet into or from the secure network, or to *deny*, which causes the packet to be discarded. The packet is accepted or denied by a specific rule if the packet header information matches all the network fields of this rule. Otherwise, the following rule is examined and the process is repeated until a matching rule is found or the default policy action is performed.

The rules that govern the firewall only match some traffic and leave the question of what to do with the rest. There are two approaches to decide on undefined traffic: default-accept and default-deny. While the default-accept approach lets all undefined traffic go through, the default-deny approach blocks all undefined traffic. If an event is unexpected, it is safer to assume that it is dangerous, at least until it has been investigated. Firewalls should therefore use the default-deny approach to block all traffic that they are not known to accept. Inevitably, this sometimes stops new or legitimate traffic; however, this inconvenience is much less painful to resolve than the alternative of allowing hostile traffic. In this thesis, we assume a default-deny policy action.

Managing firewall policies is an extremely complex task because the large number of interacting rules in single or distributed firewalls can cause significant incidents of policy misconfiguration and network vulnerabilities. Moreover, due to low-level representation of firewall rules, the semantics of firewall policies can become incomprehensible, which makes inspecting firewall policy's properties a difficult and error-prone task.

In this thesis, a visualization technique is proposed to visualize firewall rules and policies to efficiently enhance the understanding and inspection of firewall configuration. This is implemented in a tool we have devised called PolicyVis. Our tool helps the user to answer general questions such as

''Does this policy satisfy my connection/security requirements''. If not, the user can detect all misconfigurations in the firewall policy.

1.4 Introduction to Intrusion Detection and Prevention Systems

Intrusion Detection System (IDS) is the generic term given to any hardware, software, or combination of the two that monitors a system or network of systems looking for suspicious activity [10]. An IDS is used to detect different kinds of malicious behaviors and attacks that can compromise the security and trust of a computer system. Alerts raised by an IDS allow the system admin, in a swift manner, to take the appropriate actions, such as applying patches to the system or blocking the attack by setting firewalls accordingly, in order to minimize possible damages caused by attackers.

An IDS can be either a signature-based IDS or an anomaly-based IDS based on the methodology used by the engine to generate alerts. While anomaly-based IDSs detect intrusion by monitoring system activity and classifying it as either normal or anomalous, signature-based IDSs detect intrusion by matching system activities against a set of defined signatures (rules).

Moreover, based on the type and location of an IDS, it can be categorized as either a network-based IDS (NIDS) or a host-based IDS (HIDS). While NIDSs detect intrusions by examining packets that travel on network links, HIDSs detect intrusions by monitoring file system modifications, application execution logs, system calls, etc.

Intrusion Prevention System (IPS) is considered by some to be an extension of IDS. An IPS can react, in real-time, to block or prevent malicious or unwanted activities. For example, a host-based IPS can block some types of system calls and a network-based IPS can drop any packet that contains a specific string.

In this thesis, we are interested in studying signature-based NIDSs, especially policy misconfigurations in these systems. More specifically, Snort [33], the most popular open-source NIDS, is chosen as the source for this study. Besides investigating various policy misconfigurations of Snort, we focus on the misconfigurations of the *flowbits* option, which is one of the most important features that offer a stateful signature-based NIDS. We particularly concentrate on a class of *flowbits* misconfigurations that makes Snort susceptible to false negatives. We propose a method to detect the *flowbits* misconfiguration, and then suggest practical solutions with controllable false positives to fix the misconfiguration, and finally, we formally prove that the solutions are complete and sound.

1.5 Thesis Organization

This thesis is organized as follows: we describe PolicyVis in Chapter 2. In Chapter 3, we investigate Snort rules misconfiguations. Related work is discussed in Chapter 4. Finally, we show conclusions and plans for future work in Chapter 5.

Chapter 2
PolicyVis: Firewall Security Policy Visualization and Inspection

2.1 Motivation

Firewalls can be deployed to secure one network from another. However, firewalls can be ineffective in protecting networks if policies are not managed correctly and efficiently. It is crucial to have policy management techniques and tools that users can use to examine, refine and verify the correctness of written firewall filtering rules in order to increase the effectiveness of firewall security.

It is true that humans are well adapted to capture data essences and patterns when presented in a way that is visually appealing. This truth promotes visualization on data, on which the analysis is very hard or ineffective to carry out because of its huge volume and complexity. The amount of data that can be processed and analyzed has never been greater, and continues to grow rapidly.

With the necessity of guaranteeing a correct firewall behavior, users need to recognize and fix firewall misconfigurations in a swift manner. However, the complexity of dealing with firewall policies is they are attributed to the large number of rules, rules complexity and rules dependency. Those facts motivate a tool which visualizes all firewall rules in such a way that rule interactions are easily grasped and analyzed in order to come up with an opportune solution to any firewall security breach.

In this chapter, we present PolicyVis, a useful tool in visualizing firewall policies. We describe design principles, implementations and application examples of PolicyVis. We demonstrate how PolicyVis is used to discover firewall policy's properties and rule anomalies (for single and distributed firewalls).

Although network security visualization has been given strong attention in the research community, the emphasis was mostly on the network traffic [8] [18]. On the other hand, tools in [25] [38] visualize some firewall aspects, but don't give users a thorough look at firewall policies.

This chapter is organized as follows: in the next section, we describe PolicyVis design principles followed by descriptions of scenarios that show the usefulness of PolicyVis. Next, we show how rule anomalies are visualized by PolicyVis and demonstrate some examples of determining rule anomalies by using PolicyVis. We then describe visualization of distributed firewalls in PolicyVis followed by a discussion of the implementation and evaluation of PolicyVis.

2.2 PolicyVis Objectives and Design Principles

2.2.1 Objectives

PolicyVis is a visualization tool of firewall policies which helps users to achieve the following goals in an effective and fast fashion:

Visualizing rule conditions, address space and action: a firewall policy is attributed by rules format, rules dependency and matching semantics. Comprehensive visualization of firewall policies requires a mechanism of transforming firewall rules to visual elements which significantly enhance the investigation of policies. PolicyVis effectively visualizes all firewall rule core elements: conditions, address space and action.

Firewall policy semantic discovery: it is a very normal demand of users to know all possible behaviors of a firewall to its intended protected system. With advantages of visualization and many graphic options supported by PolicyVis, all potential firewall behaviors can be easily discovered, which are normally very hard to grasp in a usual context.

Firewall policy rule conflict discovery: PolicyVis can be able to not only give users a view on normal rule interactions, but also pinpoint all possible rule anomalies in the policy. This is a crucial feature of PolicyVis to become a very helpful tool for users. All kind of rule conflicts can be efficiently visualized without worrying about running any algorithm to find potential rule conflicts.

Firewall policy inspection based on users' intention: with a policy of thousands of rules, it is much likely that the user will make configuration mistakes (not rule conflicts mentioned above) in the policy which causes the firewall to function incorrectly. PolicyVis brings all firewall rules to a graphic view so that all configuration mistakes are highlighted without any difficulty.

Visualizing distributed firewalls: distributed firewalls security is as important as a single firewall, besides visualizing a single firewall; PolicyVis also lets users visualize distributed firewalls with the same efficiency in all goals mentioned above.

2.2.2 Design Principles

The fundamental design requirements for PolicyVis included:

Simplicity: It should be fairly intuitive for users to inspect firewall policies in a 2D graph using multiple fields. We chose to compress firewall rules into 2D graph with 3 factors because it is much likely that a certain field (like source port) can be ignored or not important when investigating the

policy. 2D graph is simple but quite effective in terms of helping users thoroughly look into the policy's behaviors.

Expressiveness: It is very important that users can easily capture true rule interactions so that appropriate actions can be taken immediately. PolicyVis supports very detailed and thorough visualization of all possible firewall rules' behaviors by considering all rules fields, rule orders as well as all rule actions.

Flexible Visualization scope: PolicyVis allows users to visualize what they are interested in (the target: any rule field) so that all possible aspects of the policy can be viewed and analyzed. Moreover, with multiple dimensions support, PolicyVis is flexible in letting users choose desired fields for the graph coordinates, which is convenient and effective to observe and investigate the policy from different views. Besides, there are choices on type of traffic (accepted, denied or both) which can be viewed separately to meet users' different purposes.

Compress, Focus and Zoom: It is a normal thing to take a closer look at a specific set of rules when investigating the policy. PolicyVis supports zooming so that users can closely investigate a set of considered rules. This zooming feature is very useful if too many rules get involved in the investigation and the axes get crowded. In addition, PolicyVis gives users the ability to investigate rule anomalies existing in the policy through the focusing feature. With PolicyVis, users can also visualize the whole policy at once as well as portions of the policy partitioned by ranges of a specific field. This is a helpful feature of PolicyVis when users want to consider the policy applied to a subnet or a desired portion of the network.

Using Policy segmentation: In order to investigate accepted or denied traffic only, policy segmentation with BDDs technique [1] is a powerful means employed by PolicyVis to increase the effectiveness and correctness of extracting useful information from the policy.

Using symbols, colors, notations: Policies are attributed by rules format, rules dependency and matching semantics (rule order). Moreover, firewall rules contain conditions (protocol, port and address), values (specific and wildcard) and actions (allowed and denied). PolicyVis visualizes those features using colors, symbols, and notations which are essential for users to capture quickly and easily the inside interactions and performance of firewall policies.

2.3 Multi-level Visualizing of Firewall policies

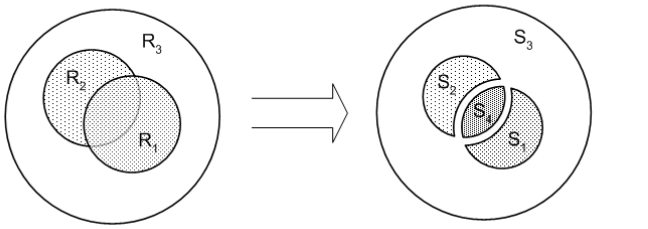
Using PolicyVis, multi-level visualizing of firewall policies can be accomplished effectively. With PolicyVis' many flexible features, users can inspect the firewall policy from different views (like port

level, address level,..etc) to understand all potential inside behaviors of the policy. In order to achieve this goal, PolicyVis deploys many methods and techniques which efficiently bring firewall policies to expressive visual views.

2.3.1 Using BDDs to segment policy and find accepted and denied spaces

Firewall policy segmentation using Binary Decision Diagram or BDD was first introduced in [1] [19] to enhance the firewall validation and testing procedures. As defined in [1], a *segment* is a subset of the total traffic address space that belongs to one or more rules, such that each of the member elements (*i.e.*, header tuples) conforms to exactly the same set of policy rules. Rules and address spaces are represented as Boolean expressions and BDD is used as a standard canonical method to represent Boolean expressions. By taking advantages of BDD's properties, firewall rules are effectively segmented into disjointed segments each of which belong to either accepted or denied space.

R1: tcp	121.63.*.*: any	143.91.78.*: any	accept
R2: tcp	121.63.71.*: any	143.91.*.*: any	accept
R3: any	*.*.*.*: any	*.*.*.*: any	deny



(a) Policy address-space (b) Segmented address-space

Table 2.1: Example of Firewall Policy Segmentation

In specific, the authors in [19] suggest constructing a Boolean expression for a policy P_a using the rule constraints as follows:

$$P_a = \bigvee_{i \in index(a)} (\neg C_1 \wedge \neg C_2 \wedge \ldots \wedge \neg C_{i-1} \wedge C_i)$$

where *index*(*a*) is the set of indices of rules that have *a* as their action and C_i is the rule condition of conjunctive fields. In other words,

$$index(a) = \{i \mid R_i = C_i \rightsquigarrow a\}$$

This formula can be understood as saying that a packet will trigger action a if it satisfies the condition C_i for some rule R_i with action a, provided that the packet does not match the condition of any prior rule in the policy. Table 2.1 shows an example of a policy of three intersecting rules forming total of four independent segments of policy address space.

PolicyVis allows users to visualize only accepted or denied traffic; therefore it is important to efficiently extract those spaces from the policy. A naïve algorithm to achieve this might take exponential running time. Fortunately, policy segmentation using BDD is quite effective in doing this. We decided to employ BDDs for segmenting rules to quickly retrieve correct accepted and denied spaces. This makes PolicyVis a reliable and fast tool. Policy rules are segmented using BDD right after they are read from the input file. This ahead-of-time rule segmentation speeds up the process when the user chooses to visualize only accepted or denied traffic.

2.3.2 Firewall Visualization Techniques

In this section, we describe the visualization techniques and methods used in our PolicyVis tool to achieve the objectives. More specific techniques and algorithms to visualize firewall anomalies are described in the "Rule Anomaly Visualization Methodology and Algorithm" section.

To achieve the visualization effectiveness, PolicyVis supports both policy segments and policy rules visualization, which depends on properties of the policy users want to examine. When dealing with only accepted or denied space, PolicyVis visualizes policy segments obtained from using BDD as mentioned in the "Using BDDs to segment policy and find accepted and denied spaces" section. However, when users choose to investigate both accepted and denied spaces together, PolicyVis visualizes policy rules because the union of both spaces returns to the original rules. Moreover, visualizing policy rules in this case helps users capture all possible rule interactions which is hard to conceive by looking at separate visualizations of both spaces.

When users investigate a firewall policy scope (a field and a value), PolicyVis collects all rules (or segments) that have the corresponding field as a superset of the scope input and visualizes those rules (or segments). When choosing a scope to investigate, users want to inspect how the firewall policy applies to that scope, thus rules (or segments) that include only the address space of the target scope. Rules (or segments) are represented as rectangles with different colors to illustrate different kinds of traffic (accepted or denied). Those colors are set transparent so that rules overlapping with the same or different actions can be effectively recognized. Moreover, different symbols (small square and circle) placed at the corner of rectangles are used for different traffic protocols (e.g., TCP, UDP,

ICMP, IGMP) and notations (i.e., tooltips or legends) are used to determine rules' order and related things.

When multiple rectangles (rules or segments) are sketched from the same coordinates, colors and symbols might not be enough to tell what kind of traffic or protocol a rectangle belongs to. Additional notations are used to clearly indicate those properties. Round brackets are used to tell if a rectangle represents denied traffic, otherwise it represents allowed traffic. Curly brackets are used to denote UDP protocol, otherwise it is TCP protocol.

PolicyVis uses three different rule fields to build the policy graph, two of which are used as the graph's vertical and horizontal coordinates and the third field is integrated into the visualization objects (e.g., at the corner of rectangles) avoiding 3D graphs for simplicity. In general, by default, PolicyVis chooses the investigated scope as one of the coordinates (axes), and from 3 remaining fields, the least common field (discussed in the "Rule Anomaly Visualization Methodology and Algorithm" section) will be the other coordinate and the second least common field will be the last dimension.

Besides, PolicyVis places rule field values along x-axis and y-axis in such a way that the aggregated values (e.g, wildcards) precedes the discrete values in the axis, or closer to the origin of the graph. Moreover, the width, the length and the position of a rectangle are chosen based on its corresponding rule's attributes so that an aggregated rule or segment (represented by a rectangle) contains its subset ones in the graph and disjoint segments or rules are represented by non-overlapping rectangles (there are no adjacent rectangles and each rectangle covers only the rule field values of its x-axis and y-axis).

2.3.3 Case Studies

In this section, we created application scenarios to explore the potential of PolicyVis to help users find the policy misconfigurations. All the scenarios were created based on the single firewall policy shown in Figure 2.1.

Order	Prot	SrcIP	SrcPort	DestIP	DestPort	Action
1	tcp	140.192.37.2	*	161.120.33.*	1433	ACCEPT
2	tcp	140.192.37.1	*	161.120.33.41	22	DENY
3	tcp	140.192.37.*	*	161.120.33.41	22	ACCEPT
4	tcp	140.192.37.4	*	161.120.33.44	1433	ACCEPT
5	tcp	140.192.37.5	*	161.120.33.44	1433	ACCEPT
6	tcp	140.192.37.*	*	161.120.33.44	1433	DENY
7	tcp	140.192.38.3	*	161.120.33.44	22	DENY
8	tcp	140.192.38.8	*	161.120.33.44	22	DENY
9	tcp	140.192.38.*	*	161.120.33.44	22	ACCEPT
10	tcp	140.192.36.2	*	161.120.34.45	22	DENY
11	tcp	140.192.36.*	*	161.120.34.45	22	ACCEPT
12	tcp	141.192.36.*	*	161.121.33.*	23	DENY
13	tcp	141.192.*.*	*	161.121.33.*	23	ACCEPT
14	tcp	141.192.37.3	*	161.121.34.3	80	DENY
15	tcp	141.192.37.5	*	161.121.34.3	80	DENY
16	tcp	141.192.37.*	*	161.121.34.3	80	ACCEPT
17	tcp	141.193.38.*	*	161.121.34.3	21	ACCEPT
18	tcp	141.193.39.*	*	161.121.34.3	21	ACCEPT
19	tcp	141.192.*.*	*	161.121.34.4	21	ACCEPT
20	tcp	141.*.*.*	*	161.121.34.5	25	ACCEPT
21	udp	142.192.*.*	*	161.122.33.43	69	ACCEPT
22	udp	143.192.*.*	*	161.122.33.43	69	DENY
23	udp	144.*.*.*	*	161.122.33.43	69	ACCEPT
24	udp	145.*.*.*	*	161.122.33.43	69	ACCEPT

Figure 2.1 : A single firewall policy

Figure 2.2. Allowed traffic to port 22

Scenario 1: The admin receives an email from the SSH server development team mentioning that there currently exists a SSH server zero-day exploit in the wild. He wants to investigate the firewall policy for accepted traffic to port *22*. The admin performs this investigation by choosing the target (scope): *Destination Port* with *22* as the input and viewing allowed traffic only as shown in Figure 2.2.

Observation: policy segments that allow traffic to SSH (port *22*) are extracted and visualized by PolicyVis as shown in Figure 2.2. Thus, the admin can then decide to block this traffic temporarily.

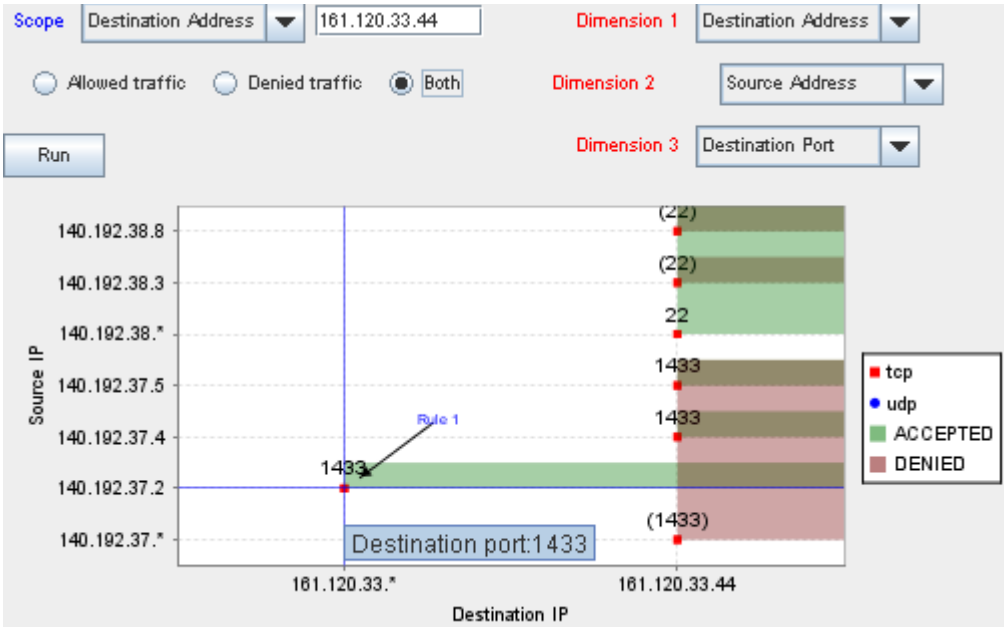


Figure 2.3. Traffic blocked and allowed to 161.120.33.44

Scenario 2: The University's student database is stolen and the database server with IP address *161.120.33.44* (possibly compromised) is suspected that it is not protected well by the firewall. The admin wants to investigate the firewall policy applied to this server. He looks into the traffic allowed and blocked by the firewall for this IP address by choosing the target (scope): *Destination Address* with *161.120.33.44* as the input as shown in Figure 2.3.

Observation: denied and allowed traffic to port *1433* (Microsoft SQL server) controlled by the firewall is almost like what the admin expected except the traffic from source address *140.192.37.2*

(from rule number *1*) which should not be allowed. The problem is traffic allowed to *161.120.33.** from rule *1* is also allowed to *161.120.33.44*. Thus, the admin might remove or change Rule *1* from the firewall.

Scenario 3: The University's whole network is down because of a denial of service attack. The admin suspects that this attack is from a specific region in a country with network IP address starting with *141.*.*.** aiming at many services including telnet, web, ftp, etc... He needs to revise the firewall policy for any traffic from any IP address starting with *141.*.*.**. The admin chooses the target (scope): *Source Address* with *141.*.*.** as the input and selects *Destination Port* (corresponding to University's network services) as one of the graph dimensions as shown in Figure 2.4.

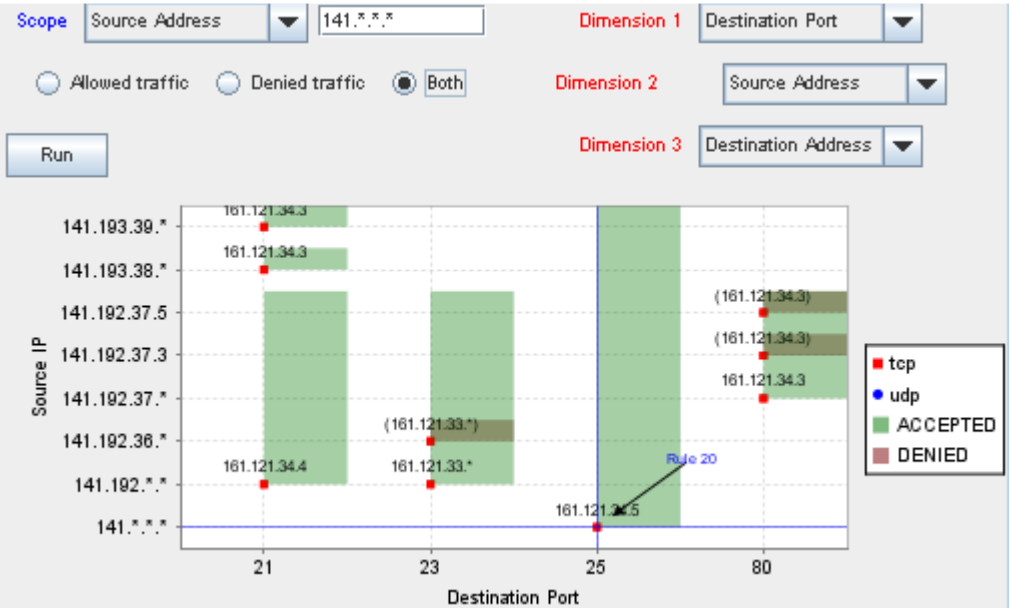


Figure 2.4. Controlled traffic from 141.*.*.*

Observation: the firewall policy currently blocks traffic to telnet service (port *23*) and web service (port *80*) from some IP addresses starting with *141*, however, SMTP service (port *25*) and FTP service (port *21*) are accessible from most of IP addresses starting with *141* and hence vulnerable to the attack. Thus, the admin may set firewall rules to block traffic from some or all addresses starting with *141* to those services as well.

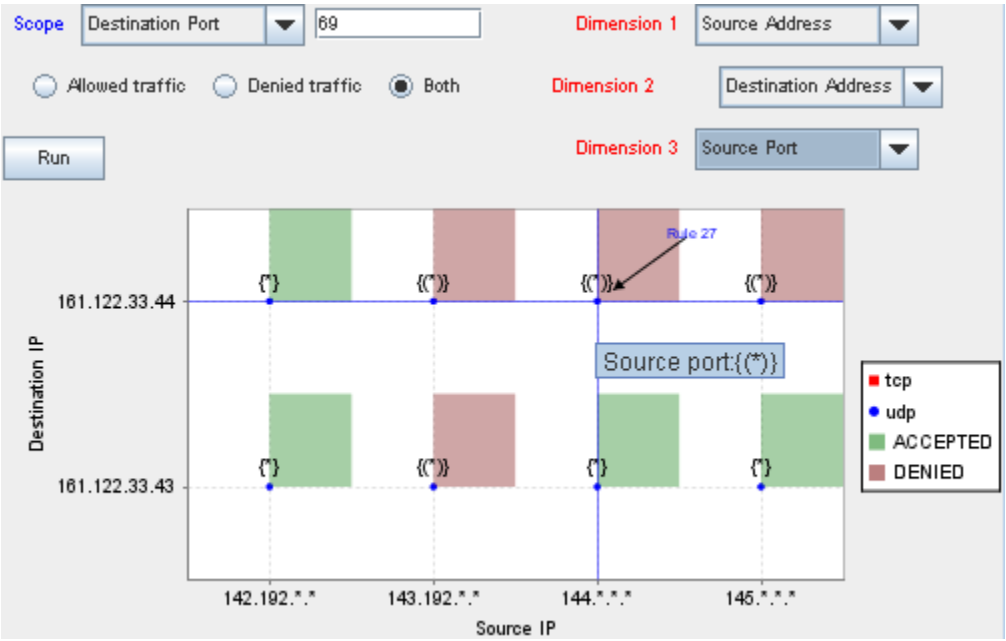


Figure 2.5. Firewall policy for destination port 69

Scenario 4: The University maintains two replicated TFTP servers (port *69*) with IP addresses *161.122.33.43* and *161.122.33.44* to satisfy students' high demand of downloading video lectures and also increase the downloading speed. However, several students still complain about low downloading speed and sometimes they are blocked from downloading. The admin first checks the two servers and sees that they both are working well. He suspects that he might make mistakes when writing firewall rules for the two servers so that one of them might not function as wanted. He needs to check the firewall policy and expects that the policy for both servers should be the same because they are replicated and have the same mission. The admin chooses the target (scope): *Destination Port* with *69* as the input as shown in Figure 2.5.

Observation: traffic controlled by the firewall to the two servers is not the same. The admin recognizes that he made mistakes blocking traffic from *144.*.*.** and *145.*.*.** to server *161.122.33.44* when they should be allowed as to server *161.122.33.43*. Thus, the admin corrects his mistakes by changing the actions in the corresponding rules in the firewall.

2.4 Visualizing Rule Anomalies

2.4.1 Definition

In this section, we mention crucial definitions and concepts of firewall policy anomalies introduced in [15] so that readers can understand how PolicyVis visualizes rule anomalies described in the "Rule Anomaly Visualization Methodology and Algorithm" section.

A firewall policy conflict is defined as the existence of two or more filtering rules that may match the same packet or the existence of a rule that can never match any packet on the network paths that cross the firewall.

Shadowing anomaly: a rule is shadowed when a previous rule matches all the packets that match this rule and they have different actions. The shadowed rule will never be activated.

Generalization anomaly: a rule is a generalization of a preceding rule if they have different actions and if the second rule can match all the packets that match the first rule.

Redundancy anomaly: a redundant rule performs the same action on the same packets as another rule such that if the redundant rule is removed, the security policy will not be affected.

Correlation anomaly: two rules are correlated if they have different filtering actions, and the first rule matches some packets that match the second rule and the second rule matches some packets that match the first rule.

2.4.2 Rule Anomaly Visualization Methodology and Algorithm

As the number of firewall rules increases, it is very likely that an anomaly will exist in the policy which threatens the firewall's security. Anomaly discovery is necessary in order to ensure the firewall's concreteness. Firewall policy advisor [15] is the first tool to discover anomalies in a firewall policy. However, it is not as expressive as PolicyVis in anomaly discovery and does not give users a visual view on how an anomaly occurs.

Four classes of firewall policy anomalies mentioned previously are visualized by PolicyVis. These anomalies are easily pinpointed by overlapping areas on the graph because an overlapping area represents for rules with overlapping traffic, which can potentially cause firewall policy anomalies. Each of the anomalies has specific features that are easily recognized on the PolicyVis graph because its corresponding overlapping area is formed (or look) differently in terms of rectangles position, colors and notations. These features are different for all four anomalies.

As PolicyVis visualizes rules in 2D-graph which shows users only 3 fields on the graph, an overlapping traffic area is a feature of a potential anomaly, however, it sometimes does not indicate that the corresponding rules are really overlapping because their 4^{th} field might be different. Nonetheless, PolicyVis still lets users visualize real anomalies by allowing related rules to be investigated more closely. When the user wants to investigate an overlapping area, he simply clicks on it and PolicyVis will focus on more details of the related rules.

PolicyVis first collects all rules containing the selected area, and then sketches a different graph for these rules. In order to correctly view real anomalies with only 3 fields used on the graph, PolicyVis needs to choose a left-out field which is the same for all the related rules. This common field is guaranteed to exist because related rules from an overlapping area must have at least 2 fields in common. PolicyVis selects the most common and least important field to be the left-out one if there are multiple common fields among the related rules.

Moreover, among three fields used for the focusing graph, PolicyVis picks the most common field over the related rules to be the third coordinate (the one integrated into visualization objects), and chooses the other two fields as the graph normal coordinates (used for axes). This coordinate selection technique assures users that, from this focusing view, an overlapping area definitely indicates at least one anomaly in the policy.

To find the most common field over some firewall rules, for each rule field excluding the *Action* field, PolicyVis needs to find a rule's field value which is a subset of all other rules' field values, and compute the number of rules that have the field value equal to that rule's field value. The field that has the biggest number is the most common field over the rules. The algorithm *FindMostCommonField* to find the most common field is implemented as shown in Table 2.2.

Algorithm FindMostCommonField
Input: *rules*
Output: *the most common field among the input rules*

```
 1:  for each field in rule.fields/{action}
 2:      if field = dest_ip or field = src_ip
 3:          C_field = *.*.*.*
 4:      end if
 5:      if field = dest_port or field = src_port
 6:          C_field = *
 7:      end if
 8:      for each rule i in rules {find a field value which is a subset of all other field values}
 9:          C_field = C_field ∩ R_i.field
10:      end for
11:      N_field = 0
12:      for each rule i in rules {count the number of rules having the field value equal to
the common subset value}
13:        if R_i.field = C_field
14:          N_field = N_field + 1
15:        end if
16:      end for
17:  end for
18:  N = max(N_dest_ip , N_scr_ip , N_dest_port , N_scr_port) {choose the most common field}
19:  if N = N_src_port
20:     return src_port
21:  end if
22:  if N = N_dest_port
23:      return dest_port
24:  end if
25:  if N = N_src_ip
26:      return src_ip
27:  end if
28:  if N = N_dest_ip
29:      return dest_ip
30:  end if
```

Table 2.2: Algorithm to find the most common field

How to recognize anomalies in PolicyVis:

The rules order is an important factor in understating the policy semantic and determining the firewall anomaly types, especially between shadowing and generalization anomalies. Besides allowing users to see the rules order by moving the mouse over the overlapping area, PolicyVis also uses surrounding rectangles (not color-filled) around the overlapping rule rectangles only in the focusing view to visualize rules or rectangles order in each overlapping area. The width (and height) difference

between a rule rectangle and its surrounding one in an overlapping area is called *boarder* and it basically shows the rule order: the rule or rectangle with bigger boarder comes first in the policy. This technique will offer an easy way to determine the type of the anomaly visually and without any manual investigation.

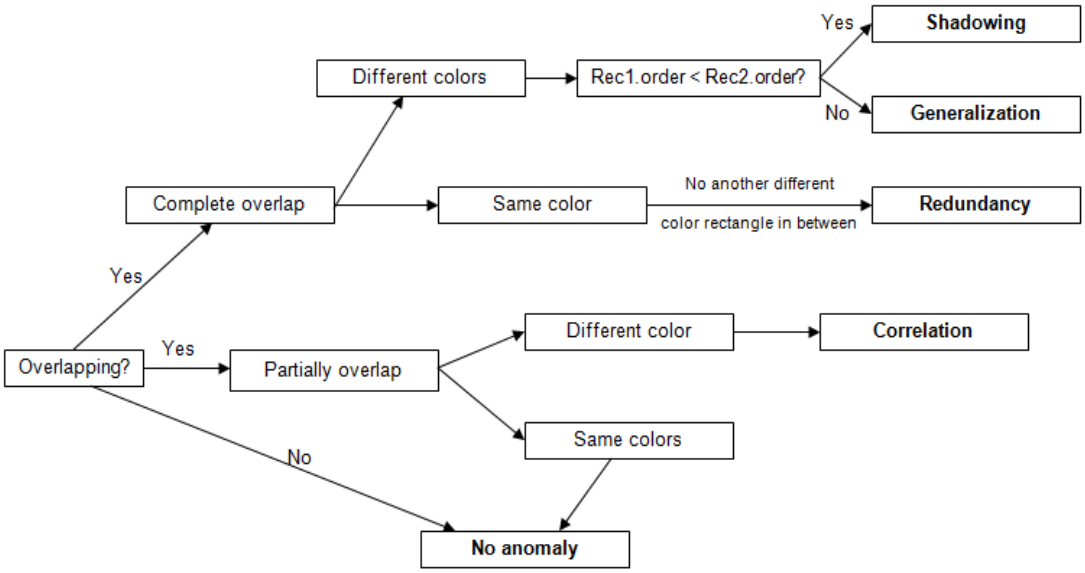


Figure 2.6. Diagram to determine possible anomalies

Shadowing and generalization anomalies: These two anomalies can be recognized by a rectangle totally contained in another rectangle but have different colors (different filtering actions), the rules order (based on extra rectangles) will decide which anomaly the overlapping area belongs to.

Redundancy anomaly: The features used to recognize this anomaly are almost the same as features used to pinpoint shadowing and generalization anomalies. Instead of having different colors, the overlapping rectangles should have the same color (same filtering action) and there is no another different color rectangle appears between them.

Correlation anomaly: This anomaly is corresponding to two rectangles with different colors partially contained in each other.

If two rectangles are not overlapping, there is no anomaly between two rules represented by those two rectangles. With the help of PolicyVis, it is straightforward to pinpoint all anomalies that might exist in the firewall policy. Figure 2.6 summarizes the method to determine different rule anomalies which is very effective in a visualized environment like PolicyVis.

2.4.3 A Case Study

Using PolicyVis to investigate the firewall policy shown in Figure 2.7, the firewall rules are visualized as shown in Figure 2.8. The admin sees many overlapping areas which might contain potential rule anomalies.

There are five suspected overlapping areas (numbered on the graph) which the user believes contain rule anomalies. From this view only, he suspects that:

1. *potential of shadowing anomaly*
2. *potential of generalization anomaly*
3. *potential of correlation anomaly*
4. *potential of redundancy anomaly*
5. *potential of generalization anomaly*

Order	Protocol	SrcIP	SrcPort	DestIP	DestPort	Action
1	tcp	140.192.37.2	*	161.120.33.*	20	ACCEPT
2	tcp	140.192.37.*	*	161.120.33.42	20	DENY
3	tcp	140.192.37.*	*	161.120.33.41	25	ACCEPT
4	tcp	140.192.37.1	*	161.120.33.41	25	DENY
5	tcp	140.192.37.3	*	161.120.33.43	21	ACCEPT
6	tcp	140.192.37.*	*	161.120.33.43	21	DENY
7	tcp	140.192.37.*	*	161.120.33.44	53	ACCEPT
8	tcp	140.192.37.4	*	161.120.33.44	53	ACCEPT
9	tcp	140.192.37.5	*	161.120.33.44	23	ACCEPT
10	tcp	140.192.37.5	*	161.120.33.44	23	DENY
11	tcp	140.192.37.5	34	161.120.33.45	22	ACCEPT
12	tcp	140.192.37.*	35	161.120.33.45	22	DENY
13	tcp	140.192.37.1	*	161.120.33.42	20	DENY
14	udp	*.*.*.*	30	161.120.33.43	50	ACCEPT
15	udp	140.192.37.*	30	161.120.33.43	50	DENY

Figure 2.7. An example of a firewall policy

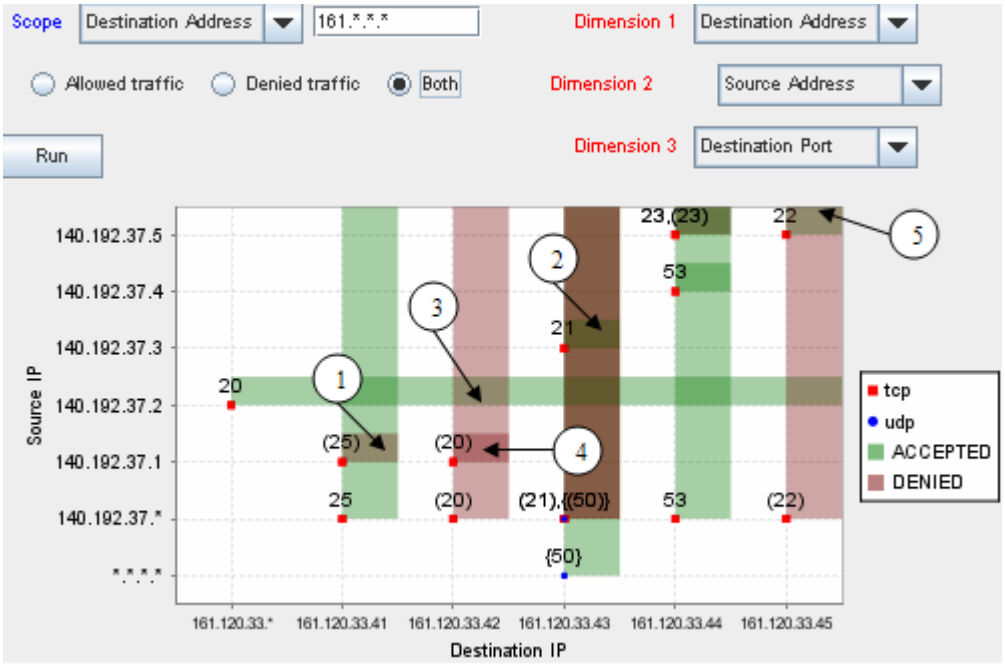


Figure 2.8. Many potential anomalies in the policy (these arrows are manually inserted)

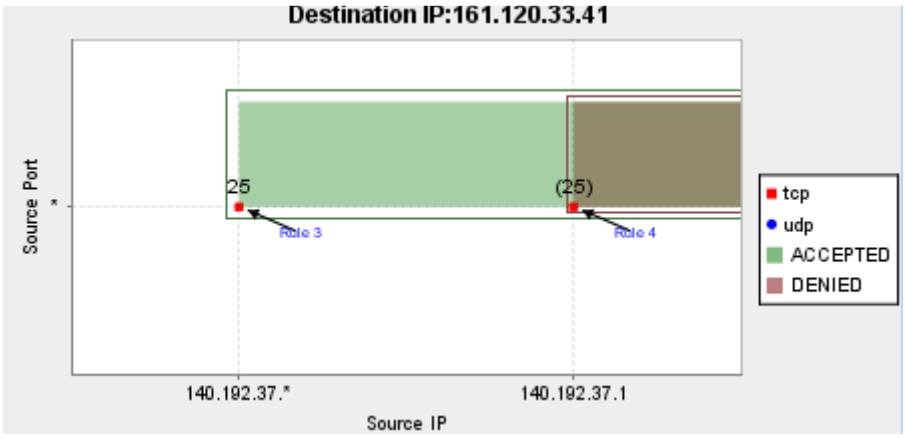


Figure 2.9. Shadowing anomaly between rule 3 and rule 4 (area number 1 from Figure 2.8)

However, in order to make sure that those anomalies are real anomalies, the admin needs to closely investigate each overlapping area. To do this, the admin simply clicks on each selected overlapping area and PolicyVis will focus on and show a more elaborated view for that area.

Shadowing anomaly visualization: When the admin clicks on the overlapping area number *1* (Figure 2.8), he is brought to the view where all traffic has the same Destination IP address *161.120.33.41* as shown in Figure 2.9. From this view, it is clear that there is a shadowing anomaly between rule *3* and rule *4* (rule *4* is shadowed by rule *3*) because the rectangle representing rule *4* is totally contained in the rectangle representing rule *3* and they have different colors. "Rule 3" and "Rule 4" tooltips appear in this case because the admin moves the mouse over the overlapping area. Without these tooltips, the admin can still tell that this is a shadowing anomaly because he knows the outer rectangle comes first in the policy based on the surrounding rectangles.

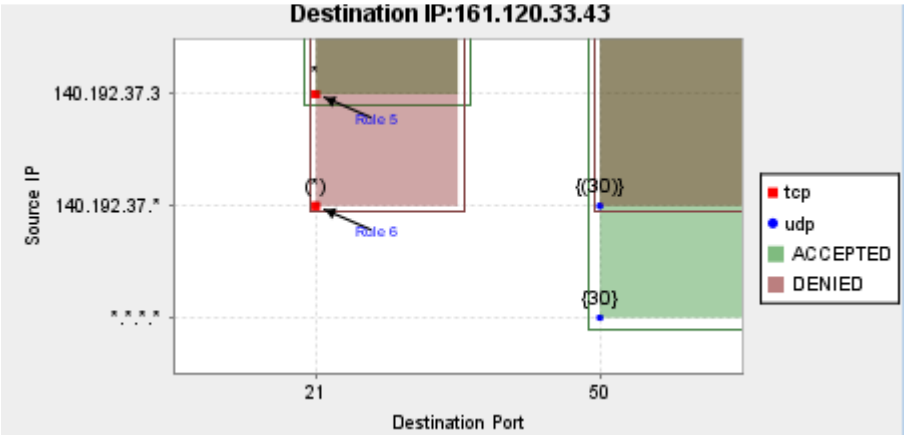


Figure 2.10. Generalization anomaly between rule 5 and rule 6 (area number 2 from Figure 2.8)

Generalization anomaly visualization: When the admin clicks on the overlapping area number *2* (Figure 2.8), he is brought to the view where all traffic has the same Destination IP address *161.120.33.43* as shown in Figure 2.10. From this view, it is clear that there is a generalization anomaly between rule *5* and rule *6* (rule *6* is a generalization of rule *5*) because the rectangle representing rule *5* is totally contained in the rectangle representing rule *6* and they have different colors. Moreover, without the tooltips ("Rule 5" and "Rule 6"), the admin still can tell that the inner rectangle comes first in the policy based on the surrounding rectangles and hence this is a generalization anomaly.

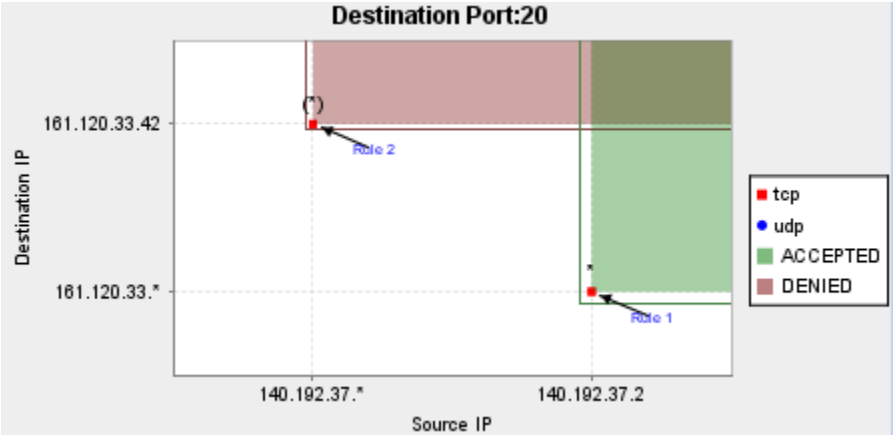


Figure 2.11. Correlation anomaly between rule 1 and rule 2 (area number 3 from Figure 2.8)

Correlation anomaly visualization: When the admin clicks on the overlapping area number *3* (Figure 2.8), he is brought to the view where all traffic has the same Destination Port *20* as shown in Figure 2.11. From this view, it is clear that there is a correlation anomaly between rule *1* and rule *2* because the rectangle representing rule *1* is partially overlapped with the rectangle representing rule *2* and they have different colors.

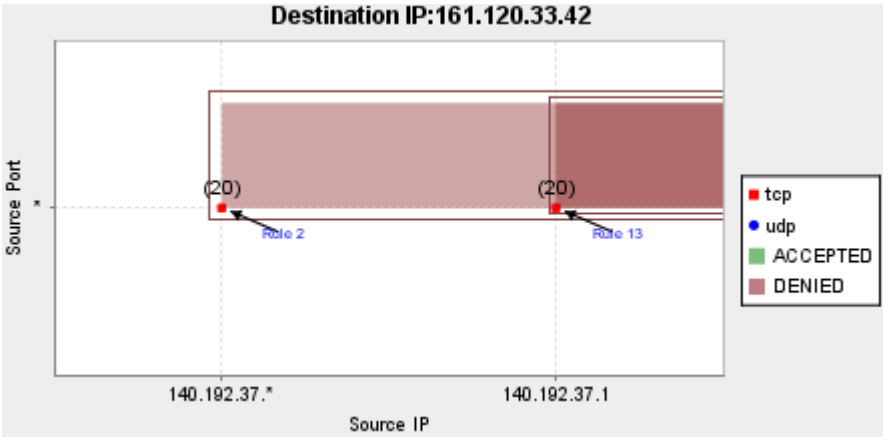


Figure 2.12. Redundancy anomaly between rule 2 and rule 13 (area number 4 from Figure 2.8)

Redundancy anomaly visualization: When the admin clicks on the overlapping area number *4* (Figure 2.8), he is brought to the view where all traffic has the same Destination IP address *161.120.33.43* as shown in Figure 2.12. From this view, it is clear that there is a redundancy anomaly between rule *2* and rule *13* (rule *13* is redundant to rule *2*) because the rectangle representing rule *13* is totally contained in the rectangle representing rule *2* and they have same color.

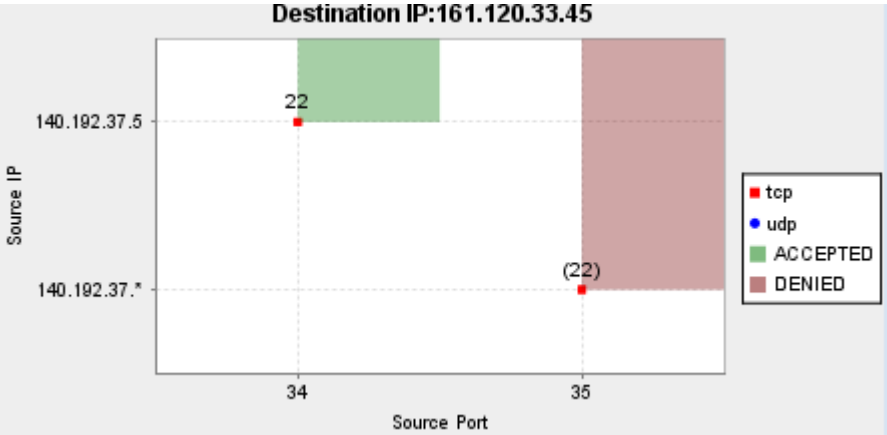


Figure 2.13. There is no anomaly in this case (area number 5 from Figure 2.8)

Overlap but no anomaly: When the admin clicks on the overlapping area number *5* (Figure 2.8), he is brought to the view where all traffic has the same Destination IP address *161.120.33.45* as shown in Figure 2.13. From this view, it is clear that there is no anomaly because the rectangles representing rules are not overlapping. Rule *11* and Rule *12* are overlapped in Fig.8 because Rule *11*'s Destination Address and Source Address are subsets of Rule *12*'s Destination Address and Source Address respectively and those 2 fields with Destination Port are chosen as dimensions for the view as shown in Figure 2.8. However, Rule *11* and Rule *12* have different Source Ports which is automatically chosen by PolicyVis as one of the dimensions for the new view as shown in Figure 2.13.

2.5 Visualizing Distributed Firewall Policy Configuration

2.5.1 Concept

While a single firewall is normally deployed to protect a single subnet or domain, distributed firewalls are essential for protecting the entire network. Any misconfiguration or conflict between distributed firewalls might cause serious flaws or damages to the network [14].

Anomalies exists not only in a single firewall but also in inter-firewalls if any two firewalls on a network path take different filtering actions on the same traffic. It is always a higher chance that distributed firewalls contain rule anomalies than a single firewall because of the decentralized property in distributed firewalls management. It is possible that each single firewall in the network might not contain any rule anomaly, but there are still anomalies between different firewalls.

Visualizing distributed firewalls gives the same benefits as visualizing single firewalls in achieving policy behavior discovery, policy correctness checking and anomaly finding. Distributed firewalls are considered as a tree where the root is the borderline firewall which directly filters traffic in and out of the network. Each node in the tree represents a single firewall which can be placed between subnets or domains in the network.

A packet from outside of the network in order to get through a firewall needs to pass all filterings of all firewalls from the root to the node representing that firewall. In the distributed firewalls view, PolicyVis creates a firewall tree based on the network topology input files and let the user pick a path (from the root to any node) he wants to examine. PolicyVis then builds up a rule set for that path by simply reading rules from nodes in order from the root to the last node. After that, PolicyVis considers this rule set as for a single firewall and visualizes it as before.

2.5.2 A Case Study

The admin wants to investigate the distributed policy configuration applied to traffic to the *Network Lab.* He first changes the view to *Distributed Firewalls* view and expands the tree to get to the *Network Lab* node. As shown in Figure 2.14, PolicyVis creates a new rule set containing all rule sets from firewalls on the path in this order: *University of Waterloo*, *Math faculty*, *CS department* and *Network Lab.*

After building up the rule set for the path from *University of Waterloo* to *Network Lab*, PolicyVis allows the admin to start visualizing the path policy. In this visualization, the admin chooses to investigate all rules on this firewall path that control traffic to any destination address in the university network by choosing the scope *Destination Address* with value *161.*.*.*.

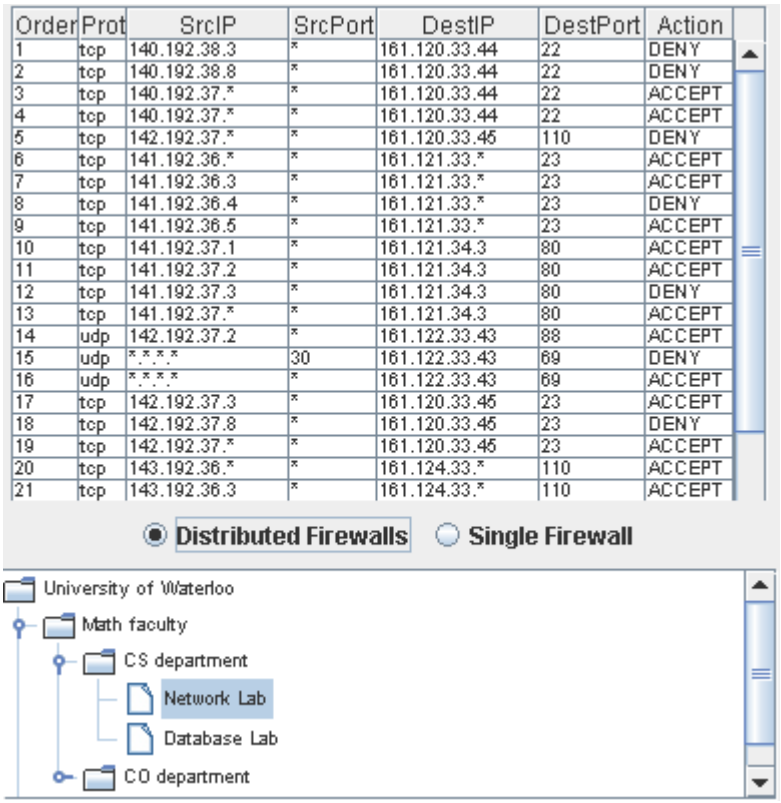


Figure 2.14. An example of distributed firewalls

In this case, there are multiple subnets getting involved because multiple firewalls are considered at once. PolicyVis not only lets the admin visualize all the subnets at the same time, but also supports a single view on each subnet and the admin can switch views between subnets easily. In this example, there are six subnets whose traffic are controlled by the firewalls on the path and the *Network Lab* subnet *161.120.33.** is currently viewed and analyzed by the admin (Figure 2.15). The admin can change the view to a different subnet by clicking on the *Next* or *Previous* button.

It is easy to recognize that while the single firewall placed at the *Network Lab* subnet (Figure 2.16) which only controls traffic to *161.120.33.** does not contain any anomaly, the distributed firewalls (Figure 2.15) seems to have anomalies (overlapping areas). In fact, there is a shadowing anomaly in this case between a rule in the *University of Waterloo* firewall and a rule in the *Network Lab* firewall.

Figure 2.15. Visualization of all firewall policies to subnet 161.120.33.*

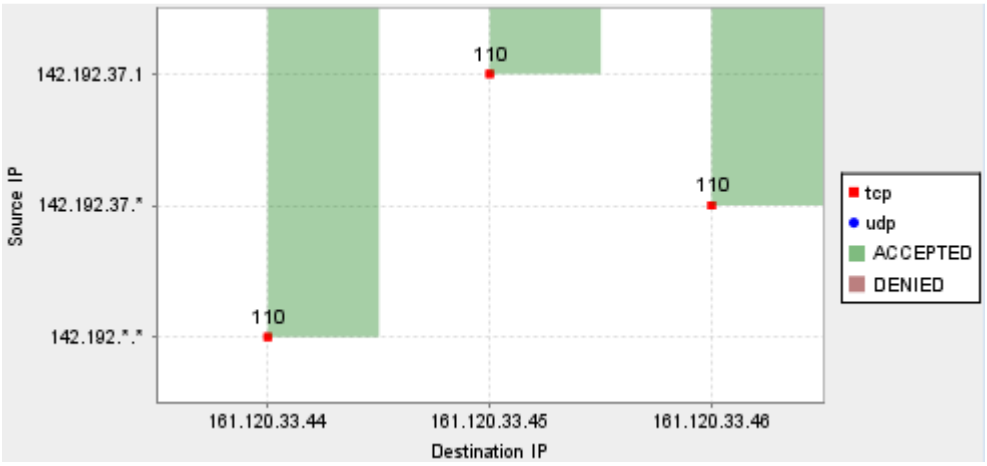


Figure 2.16. Visualization of the *Network Lab* subnet firewall

2.6 Implementation and Evaluation

We implemented PolicyVis using Java and Jfreechart [13], a free open source Java chart library, in PolicyVis to make it easy for displaying charts in the graph. We also used Buddy [26] for BDD representation of firewall policies.

In this section, we present our evaluation study of the usability and efficiency of PolicyVis. To access the practical value of PolicyVis, we not only created firewall policies randomly (with and without rule anomalies), but also used real firewall rules from our own machines for the evaluation study.

Each firewall used in the evaluation test has from 30 to 45 rules. We then asked 11 people (with varying level of expertise in the field) under test to use both PolicyVis and raw firewall rules to find some specific firewall properties (like what traffic is allowed to a chosen domain or which machine has Web accessible web traffic and so on), firewall and locate rule anomalies in the firewalls. We recorded the time to answer each task by using each method for all people and computed the average time over all.

Task\ Method	**PolicyVis**	**Raw firewall rules**
Find firewall properties	*3.12 minutes*	*10.44 minutes*
Find firewall anomalies	*1.98 minutes*	*12.78 minutes*

Table 2.3: Average estimated time to achieve each task by using each method

People in this evaluation test were getting familiar with PolicyVis very quickly and very confident with features supported by PolicyVis. As shown in Table 2.3, the average time to achieve each task by using PolicyVis is much faster than by investigating raw firewall rules, especially in finding firewall anomalies (with small standard deviation). This evaluation test demonstrated that PolicyVis is a very user-friendly tool with high usability and efficiency.

Chapter 3
Snort Policy Misconfigurations

3.1 Snort Rule Background

Snort [33] is a popular, free and open source Network Intrusion Prevention System and Network Intrusion Detection capable of performing packet logging and real-time traffic analysis on IP networks. Snort might be considered a lightweight NIDS because it has a small footprint, has relatively small requirements, does not always demand a suite of specialized servers, and runs on a variety of operating systems [10]. Even though Snort supports some anomaly detection through its pre-processors, Snort is more like a signature-based NIDS and famous for its intrusion detection capabilities that match packet contents against a set of rules. Snort rules are easy to write and Snort supports a strong and flexible rule language with a variety of options, which allow users to inspect all fields of a packet. A snort rule has two parts: rule header and rule body.

3.1.1 Rule Header

A Snort rule header contains the following options:

***Action*:** the type of action to take when the rule matches a packet. The action can be *Pass* (tell Snort to ignore the packet), *Log*(tell Snort to log the packet), *Alert* (tell Snort to send an alert message), *Activate* (create an alert and then activate another rule for checking more conditions), or *Dynamic* (*Dynamic* action rules are invoked by other rules using the *Activate* action). Besides these actions, a user can define his own action for different purposes.

***Protocol*:** the type of packet the rule matches. The protocol can be *IP, ICMP, TCP or UDP.*

***Address*:** there are two address parts in a rule header. These addresses are used to check the source from which the packet originated and the destination of the packet. The address can be a defined variable, a single *IP* address, a CIDR block or a list of these.

Port: there are two port parts in a rule header. One is for the source and the other is for the destination of the matching packet. The port can be a defined variable, a single port number, or a port range.

***Direction*:** this field determines the source and destination addresses and port numbers in the rule header. A "→" symbol means the source is on the left and the destination is on the right. A "←" symbol means the source is on the right and the destination is on the left. A "<>" symbol shows that

the rule will be applied to packets traveling on either direction. In this thesis, we assume that only "→" is used because a rule using "←" or "<>" can be rewritten (or splitted) to use the "→" direction.

3.1.2 Rule Body

The rule body begins and ends with the open and brackets "()". There are five types of options that can be used in the rule body: General rule options, Payload detection, Non-payload detection, Post-detection, and Event thresholding.

***General rule options*:** these options provide Snort with information about the rule itself or pass on information to the analyst. Examples: *msg*, *reference*, *classtype, etc.*

***Payload detection options*:** these options let users look inside the packet payload. Examples: *content, uricontent, depth, offset, distance, byte_test, etc.*

Non-Payload detection options: these options let users look for non-payload data, like inside the packet header. Examples: *dsize, flowbits, ttl, flags, etc.*

Post-Detection options: these options specify actions Snort will take if the rule is fired. Examples: *logto, tag, resp, etc.*

***Event thresholding options*:** these options can be used to reduce the number of logged alerts for noisy rules. Examples: *limit, threshold, both, etc.*

Here is an example of a Snort rule from [9]:

alert tcp $EXTERNAL_NET any -> $HTTP_SERVERS $HTTP_PORTS (msg:"BLEEDING-EDGE EXPLOIT CVSTrac filediff Arbitrary Remote Code Execution"; flow: to_server,established; uricontent:"filediff|3f|f="; nocase; pcre:"/filediff\?f=.+&v1=[\d.]+&v2=[\d.]+\;.+/Ui"; reference:bugtraq,10878; reference:cve,2004-1456; classtype:web-application-attack; sid:2002697; rev:4;)

This rule is used to detect any attempt to use *filediff* from the *CVSTrac* application to execute arbitrary code on the HTTP server.

- Rule header: *alert tcp $EXTERNAL_NET any -> $HTTP_SERVERS 80*. This rule will *alert* on *tcp* packets from source address *$EXTERNAL_NET* (this is a variable defined in *snort.conf*), source port *any* to destination address *$HTTP_SERVERS* (defined in *snort.conf),* destination port *80.* Moreover, those packets need to match the following rule body:
- Rule body: *(msg:"BLEEDING-EDGE EXPLOIT CVSTrac filediff Arbitrary Remote Code Execution"; flow:to_server,established;uricontent:"filediff|3f|f=";nocase;pcre:"/filediff\?f=.+&v1=[\d.]+&v2=[\d.]+\;.+/Ui"; reference:bugtraq,10878; reference:cve,2004-1456; classtype:web-application-attack; sid:2002697; rev:4;).* The *msg* option tells Snort to log the message inside the quotations if

the rule is triggered. The *flow:to_server, established* option tells Snort to match only packets sent from server and from an established connection. The *uricontent* option requires a packet to contain the string inside the quotations in its normalized URI (*nocase* means no case sensitive). The *pcre* option requires a packet to contain a string that matches the regular expression inside the quotations. The *reference* options show the reference source of the attack. The *classtype* option denotes the type of the attack, which is web application attack.

3.2 Snort Rule Misconfigurations

A snort policy misconfiguration is any kind of error or mistake in rule specification that causes at least one of the following problems:

Snort misbehaves: Snort does not carry out the policy exactly as the user expected. This will cause false positives or false negatives. In other words, Snort will raise an alert on benign traffic or miss real attacks.

Snort performs poorly: Snort takes longer time than it should to carry out the policy. In the worst case, Snort cannot function properly anymore or stops functioning.

In this section, we classify various Snort rule misconfigurations and give examples for each of those. However, we only consider misconfigurations based on the rules themselves and assume that all other different components related to the intrusion detection are well configured. Moreover, within the scope of this thesis, it is impossible to give solutions for all Snort rule misconfiguration classes. Instead, we deliberately concentrate on a class of *flowbits* misconfiguration that makes Snort susceptible to false negatives. We show how the *flowbits* misconfiguration can be exploited, propose a method to detect the *flowbits* misconfiguration and suggest practical solutions with controllable false positives to fix the misconfiguration.

There are two kinds of misconfigurations that can occur in a Snort policy: misconfigurations within a single rule and those amongst multiple rules.

3.2.1 Misconfigurations Within a Single Rule

- ***Rule with redundant option(s)***: A redundant option in a rule does not have any effect on how the rule matches a packet. In other words, a redundant option can be removed from the rule without

changing the rule's semantics. A rule with redundant options potentially causes false positives and false negatives. If the user mistakenly provides a redundant option and still thinks that the option contributes to the rule's semantics, it is very likely that the rule will not work as expected. Here are two examples:

Example 1: alert tcp any any -> any any (content:"shell"; offset:3; offset:4;)

The *offset* option is a modifier to the *content* option and it allows the rule writer to specify where to start searching for a pattern within a packet. There are two *offset* modifiers to the *content* option in this rule. The first *offset* (offset:3) is a redundant option and only the second *offset* (offset:4) (comes after in the rule) contributes the rule's semantics. A packet with the payload starting with "*xxx*shell" (where '*x*' is any character) does not trigger this rule.

Example 2: alert tcp any any -> any 80 (content:"shell"; http_client_body; http_uri;)

In this rule, while the *http_client_body* option restricts the search for "shell" to the normalized body of an HTTP client request, the *http_uri* option restricts the search for "shell" to the normalized request URI field. The *http_client_body* option is redundant because only the *http_uri* option (comes after in the rule) contributes to the rule's semantics. This rule does not match an HTTP request that contains "shell" in its normalized body but does not have "shell" in its normalized URI field.

- ***False rule***: A rule that has inconsistencies amongst its options' values which cause the rule always evaluated to false. This kind of rule never matches any packet. Inconsistencies can happen amongst options within the rule body, or between options in the rule header and the rule body. Here are some examples of false rules:

Example 1: alert tcp any any -> any any (content:"AA"; dsize:1;)

This rule will not match any packet because there is no packet with the payload of size 1 byte and the payload contains "AA" which is 2 bytes.

Example 2: alert tcp any any -> any any (content:"AAA"; offset:5; dsize:4;)

This rule is false because it matches any packet with the payload of size 4 bytes but expects the packet payload contain the string "AAA" from the 5th byte.

Example 3: alert tcp any any -> any any (content:"|00|"; depth:2; content:"|01|"; depth:2;)

This rule will not match any packet because the first 2 bytes of any packet payload cannot be "00" and "01" at the same time.

Example 4: alert tcp any any -> any any (isdataat:5; dsize:2;)

This rule is false because there is no packet with the payload of size 2 bytes and the payload has data at the 5^{th} byte.

Example 5: alert tcp any any -> any any (msg:"malicious matching rule"; content:"|00|"; flowbist:isset,test; *flowbits*: isnotset, test;)

This rule is false because the *flowbits* constraints are never satisfied. The label "test" cannot be set and not set at the same time.

Example 6: alert tcp 10.10.1.0 any -> 10.10.1.1 any (sameip;)

This rule will not match any packet because the option *sameip* only matches packets having the same source and destinations addresses. However, the source and destination addresses from the rule header are different.

- ***Loose rule***: A rule with simple matching options that possibly matches a lot of traffic, which potentially causes false positives. We consider a loose rule as a misconfiguration not only because it potentially causes false positives, but also because it can be exploited to bypass Snort detection (causes false negatives) when the *flowbits* option is used (discussed in Section 3.3).

- ***Slow rule***: To avoid some evasion cases, Snort must use recursion (on payload detection options) to perform pattern matching. The order of matching options (in the rule body) in a rule might affect the rule matching performance. A slow rule is defined as a rule without optimized order of matching options. A slow rule can be rewritten to achieve better matching performance, especially against unmatched packets. Here are two examples of slow rules:

Example 1: R1: alert tcp any any -> any any (content:"shell"; dsize:5;)

R2: alert tcp any any -> any any (dsize:5; content:"shell";)

R1 and R2 have the same matching semantics; however R1 is a slow rule and R2 has optimal order of matching options. While a packet with payload containing 10 "shell"s could cause R1 to perform 10 pattern match attempts and 10 *dsize* checks, R2 will stop checking this packet immediately after the *dsize* option fails to match.

Example 2: R1: alert tcp any any -> any any (content:"A"; content:"AA"; content:"AAA";)

R2: alert tcp any any -> any any (content:"AAA"; content:"AA"; content:"A");

R1 and R2 have the same matching semantics; however R1 is a slow rule and R2 has optimal order of matching options. While a packet with payload "AA" causes R1 to perform at least 3 pattern match attempts, R2 will stop checking this packet immediately after the first pattern "AAA" fails to match.

3.2.2 Misconfigurations Amongst Multiple Rules

Misconfigurations occur amongst multiple rules when the order of rule triggering affects the correctness of Snort detection. In other words, the triggering of one rule depends on the triggering of one or many other rules. In this thesis, we assume that Snort is configured with the default rule application order *activate*→ *dynamic*→*pass*→*alert*→*log*. It means that Snort will match a packet in this order of rule types.

In Snort, there are 3 ways in which the triggering of one rule might affect the triggering of another rule:

a) *Pass* rules: for *activate*, *dynamic*, *alert* and *log* rules, Snort allows multiple events trigger. It means that Snort does not stop applying rules to a packet when a match is made for these rules. However, when a packet matches a *pass* rule, it is not processed by Snort anymore. Therefore, the triggering of a *pass* rule might affect the triggering of an *alert* or *log* rule.

b) *Dynamic/Activate* rules: a *dynamic* rule is able to trigger only if its corresponding *activate* rule is triggered. Therefore, the triggering of a *dynamic* rule depends on the triggering of its corresponding *activate* rules.

c) Rules using the *flowbits* option: the *flowbits* option allows Snort to detect an intrusion over multiple rules and multiple packets. Within a *flowbits* rule set, the triggering of one rule depends on the triggering of one or many rules in the rule set.

While the triggering of a *pass* rule might affect the triggering of an *alert* or *log* rule on the same packet, the triggering affection in *dynamic/activate* rules and rules using the *flowbits* option occur on different packets, i.e., the triggering of one rule on a packet might affect the triggering of another rule on another packet. Therefore, we divide misconfigurations amongst multiple rules into two corresponding classes: misconfigurations on a single packet and misconfigurations on multiple packets.

3.2.2.1 Misconfigurations On A Single Packet

Considering misconfigurations on the same packet, Snort rules also have the same types of misconfigurations as in firewall rules [15]. However, compared with a firewall rule, a Snort rule has one more tuple, which is the rule body. A snort rule has the following format:

<action> <protocol> <src_ip> <src_port> → <dst_ip> <dst_port> <body>

A packet can be considered as a binary string. Let L_R represent all strings that match R's body. We first define all possible relations that may exist between Snort rules (independent of the rule *action*), and then we classify different misconfigurations that may exist based on these relations.

Definition 1: R_x and R_y are *disjoint* if

$\exists$ i: $R_x[i] \cap R_y[i] = \varnothing$ where i $\in$ {protocol, s_ip, s_port, d_ip, d_port, L_R}

We denote R_x R_D R_y.

Definition 2: R_x and R_y are *exactly matching* if

$\forall$ i: $R_x[i] = R_y[i]$ where i $\in$ {protocol, s_ip, s_port, d_ip, d_port, L_R}

We denote R_x R_{EM} R_y.

Definition 3: R_x and R_y are *inclusively matching* if they are not *exactly matching* and

$\forall$ i: $R_x[i] \subset R_y[i]$ or $R_x[i] = R_y[i]$ where i $\in$ {protocol, s_ip, s_port, d_ip, d_port, L_R}

R_x is called the *subset match* and R_y is called the *superset match*.

We denote R_x R_{IM} R_y.

Definition 4: R_x and R_y are *correlated* if they are neither *exactly matching* nor *inclusively matching* and

$\forall$ i: $R_x[i] \cap R_y[i] \neq \varnothing$ where i $\in$ {protocol, s_ip, s_port, d_ip, d_port, L_R}

We denote R_x R_C R_y.

It is easy to see that any two Snort rules are related by one and only one of the defined relations. Based on these relations, we classify all possible misconfiguations on the same packets as follows:

• ***Shadowing misconfiguration***: a rule is shadowed if the rule's action is *alert* or *log* and there is a *pass* rule that matches all the packets that match this rule. A shadowed rule is never triggered. Formally, rule R_y is shadowed by rule R_x if the following conditions hold:

$$R_x[\text{action}] = pass,\ R_y[\text{action}] = alert \text{ or } log,\ R_y\ R_{EM}\ R_x \text{ or } R_y\ R_{IM}\ R_x$$

Example: R1: pass tcp 192.168.0.100 → 192.168.1.109 (content:"water";)

R2: alert tcp 192.168.0.100 → 192.168.1.109 (content:"waterloo";)

In this example, R2 is shadowed by R1 because R1 matches all the packets that match R2. R2 is never triggered because all packets that match R2 will be let through by R1.

• ***Generalization misconfiguration***: an *alert* (or *log*) rule is a generalization of a *pass* rule if the *alert* (or *log*) rule matches all the packets that match the *pass* rule. Formally, rule R_x is a generalization of rule R_y if the following conditions hold:

$$R_x[\text{action}] = alert \text{ or } log,\ R_y[\text{action}] = pass,\ R_y\ R_{IM}\ R_x$$

Example: R1: alert tcp 192.168.0.100 → 192.168.1.109 (content:"water";)

R2: pass tcp 192.168.0.100 → 192.168.1.109 (content:"waterloo";)

In this example, R1 is a generalization of R2 because R1 matches all the packets that match R2. Some packets which match R1 will be dropped by R2.

• ***Redundancy misconfiguration***: a redundant rule has the same action on the same packets as another rule such that if the redundant rule is removed, the security policy will not be affected. Formally, rule R_y is redundant to rule R_x if the following conditions hold:

$$R_x[\text{action}] = R_y[\text{action}],\ R_y\ R_{EM}\ R_x \text{ or } R_y\ R_{IM}\ R_x$$

Example: R1: log tcp 192.168.0.100 -> 192.168.1.* (content:"water";)

R2: log tcp 192.168.0.100 -> 192.168.1.109 (content:"waterloo";)

In this example, R2 is redundant to R1 because both rules tell Snort to log matched packets and all packets triggering R2 will trigger R1.

• ***Correlation misconfiguration***: Two rules are correlated if one rule's action is *alert* or *log* and the other rule's action is *pass*. The first rule matches some packets that match the second rule and the second rule matches some packets that match the first rule. Formally, rule R_x and R_y are correlated if the following conditions hold:

$$R_x[\text{action}] = pass,\ R_y[\text{action}] = alert \text{ or } log,\ R_y\ R_C\ R_x$$

Example: R1: pass tcp 192.168.0.100 -> 192.168.1.109 (content:"waterloo";)

R2: alert tcp 192.168.0.100 -> 192.168.1.* (content:"university";)

In this example, R1 and R2 are correlated because they both match packets from 192.168.0.100 to 192.168.1.109 with the payload containing both "waterloo" and "university". However, R1 does not match packets from 192.168.0.100 to 192.168.1.109 with the payload "university", which match R2. On the other hand, R2 does not match packets from 192.168.0.100 to 192.168.1.109 with the payload "waterloo", which match R1.

3.2.2.2 Misconfigurations On Multiple Packets

• ***Missing corresponding rule***: *activate* and *dynamic* rules always go together as a pair. This misconfiguration occurs when there is an *activate* rule without its corresponding *dynamic* rule or vice versa.

• ***Unreachable rule***: this misconfiguration has the same effect as the shadowing misconfiguration on a single packet, i.e., a rule is never triggered (even though the rule is not a false rule). However, the cause of this misconfiguration is one of the followings:

- A *dynamic* rule is unreachable: if the corresponding *activate* rule is a false rule
- A *flowbits* rule is unreachable: each *flowbits* rule set can be represented or modeled by a DFA (more details are discussed in Section 3.3). A *flowbits* rule is unreachable if the states from which the rule can be triggered are not reachable. Either these states are not reachable from the *flowbits* rules set's DFA, or one (or more) another *flowbits* rule is shadowed (or a false rule) that causes these states unreachable.

Example: R1: alert tcp any any -> any 80 (content:"waterloo1"; flowbits:set, label1;noalert)

R2: alert tcp any any -> any 80 (content:"waterloo2"; dsize:3;flowbits:set, label2;noalert);

R3: alert tcp any 80 -> any any (content:"waterloo3"; flowbits:isset, label2; flowbits:set,label3; noalert)

R4: alert tcp any any -> any 80 (msg:"attack 1 detected"; content:"waterloo4"; flowbits:isset, label1;)

R5: alert tcp any any -> any 80 (msg:"attack 2 detected"; content:"waterloo5"; flowbits:isset, label4;)

This *flowbits* rule set has 5 rules. R5 can be triggered only if *label4* is set (*flowbits:isset, label3;*), however, there is no rule in the rule set that sets *label4*. Therefore, R4 is unreachable. Moreover, R3 can be triggered only if *label2* is set, however, R2 is the only rule that sets *label2* and R2 is a false rule. Hence, R3 is unreachable.

• ***False rule set***: this misconfiguration is similar to the ***false rule*** misconfiguration within a single rule. However, this miconfiguration is related to the detection of a rule set instead of a single rule. A pair of *activate/dynamic* rules or a *flowbits* rule set raises an alert on a sequence of packets. The set of all packet sequences that trigger an alert can be considered as the rule set's detection language (more details are discussed in Section 3.3). A rule set is false if its detection language is empty. A false rule set never raises an alert on any packet sequence.

Example: considering the rule set used as the example for the unreachable rule misconfiguration, if R4 is shadowed by a *pass* rule, then both R4 and R5 are unreachable. Because only R4 and R5 can raise an alert (other rules use the *noalert* option), the rule set therefore will not raise an alert on any packet sequence. This makes it a false rule set.

• ***Vulnerable rule set***: a *flowbits* rule set makes Snort susceptible to false negatives, i.e., allows the attacker to carry out an attack without Snort detection. This misconfiguration is studied in detail in the following section.

3.3 *Flowbits* Misconfiguration

3.3.1 *Flowbits* Background

The *flowbits* option was first introduced in Snort 2.1.1 and is most useful for TCP sessions, as it allows rules to generically track the state of an application protocol [43]. Sometimes, looking at one packet at a time, one rule at a time could not tell whether an event occurs if the event is a series of actions that we need to keep track of. In fact, many times we need to inspect more than just a single packet with more than one rule to detect an attack, especially complicated ones. Before the addition of *flowbits*, Snort could not do this. The *flowbits* detection option makes snort rules stateful and allows the detection engine to track state across multiple packets in a single session. Stateful detection is so important that sometimes it is implemented separately for a specific service [47].

With *flowbits*, we can essentially set a flag that another rule can check and take into consideration. We can think of this in terms of streams and multiple rules. We can look at flowbit usage in terms of a chain of events, or a logic flow: If condition 1 happens, set a flowbit. If this flowbit is set and you see condition 2 but not condition 3, generate an alert. That second event can occur many packets later in the stream, or seconds or minutes later in the stream [10]. The *flowbits* option works by using labels to set and change the session state. The *flowbits* option is used with this format: *flowbits*: [set|unset|toggle|isset,reset,noalert][,<LABEL>];

E.g: Here is a rule set using *flowbits* from BleedingEdge [9]:

alert tcp $EXTERNAL_NET any -> $HOME_NET 21 (msg:"BLEEDING-EDGE FTP USER login flowbit"; flow: established, to_server; content: "USER"; nocase; *flowbits: set, login; flowbits: noalert;*)
alert tcp $EXTERNAL_NET any -> $HOME_NET 21 (msg: "BLEEDING-EDGE FTP HP-UX LIST command without login"; flow: established, to_server; content: "LIST"; nocase; *flowbits: isnotset, login;*)

Table 3.1: Example of Snort *flowbits* rule set

These rules in Table 3.1 use *flowbits* to follow a FTP session. The first rule checks if a user is trying to log in the FTP server and sets the label "login". The second rule will raise an alert if the user uses the "LIST" command, however, only if the user has not logged in yet. In other words, the second rule is triggered only if the label "login" is not set. This rule set might do a good job in detecting someone using the "LIST" command without even trying to log in the server. However, an attacker can always try to log in first (even not authorized) and then use the "LIST" command. This way allows the attacker to execute the LIST command without really logging in, but Snort still thinks that

the user already logged in. As a result, the use of the "LIST" command is allowed. In this case, Snort misjudges the actual session, and of course no alert is raised.

Because of its powerfulness as well as usefulness, the *flowbits* option is now getting more attention and used more frequently. In about 20K rules we collected from many different sources on the internet, about 3K rules (15%) make use of *flowbits*. The *flowbits* concept is used in many different IDSes as well, like in Cisco IPS [11], Bro [32],etc. For example, Bro [32] allows users to write a script to raise an alert based on a sequence of events. This is equivalent to the use of *flowbits* in Snort if we consider handling each event by a rule and set *flowbits* accordingly so that Snort will raise an alert if it see the event sequence.

Complicated attacks against services with complex protocols are normally hard to detect. The *flowbits* option allows Snort to keep track of the session state and raise an alert if an event happens when the session is in a specific state. With the use of *flowbits*, Snort keeps track of the session by maintaining its own session state, called *in-snort session state*. The in-snort session state is supposed to reflect the actual session state, in other words, the *actual session state* is simulated in Snort as much as possible. However, due to the protocol complication (of the session's corresponding service) and overheads caused by doing so, it is impossible to have a complete simulation of the actual session state in Snort.

3.3.2 *Flowbits* Misconfiguration Establishment

In this thesis, we concentrate on a class of *flowbits* misconfiguration that allows the attacker to carry out an attack without Snort detection. This kind of *flowbits* misconfiguration gives the attacker a chance to make Snort misunderstand the actual session state, i.e. the attacker can force a difference between the actual and the in-snort session states (Snort thinks that the session is in a state but the session is actually in a different state) at some point in the attack, therefore, possible to bypass Snort detection. The possibility of this *flowbits* misconfiguration class is due to:

- ***Complex sessions***: A complex session may have many possible states and many possible transitions back and forth between these states. Moreover, in order to avoid false positives, a Snort policy needs to consider "innocent" paths, which Snort should not raise an alert. Complicated states, complex state transitions, and "innocent" paths might give the attacker a chance to put the session into a desired state before finishing the attack so that the attack goes unnoticed.

- ***Insecure detection approaches***: For a given attack, even a simple one, there are possibly many approaches to detect it, and more than one rule set can be created to detect the attack. Even though these rule sets semantically do the same thing (raise an alert if the attack occurs), they might make use of different session states, so it has different ways to come up with the alert. There are approaches which logically leave a path allowing the attacker to finish the attack without being detected. For example, in order to detect if a normal user (non-admin) uses a bad command in a telnet session, one straight approach is to only raise an alert if the bad command use is detected when the session is in a state that a normal user has logged in. Another approach is to always raise an alert if the use of the bad command is detected except when the session is in a state that the admin has logged in. While the first approach can always catch the attack because the attacker has to log in as a normal user before using the bad command, the second approach opens a door for the attacker: if he can pretend that he has successfully logged in as the admin, then he can execute the bad command.

This *flowbits* misconfiguration can be exploited in practice because of these factors:

- ***Packet-based property of Snort***: As we know, Snort is a packet-based NIDS and basically most packets received by Snort, after processed by preprocessors, are passed down to the detection engine to make sure all possible attacks are considered. Exploiting this feature, we can construct a packet that is not processed by the receiver's application layer but still inspected by the Snort detection engine and triggers a given rule. This will cause the inconsistency between the actual and in-snort session states.
- ***Loose rules***: A loose rule is a rule with simple matching options that possibly matches a lot of traffic. A loose rule matches a packet by only checking if its payload contains some strings and does not examine other fields and the packet structure. It is the fact that if a rule is loose, it might cause false positives. However, a loose rule used with *flowbits* might allow an attacker to carry out an attack without being detected, which causes false negatives. The reason people still write loose rules is in a normal connection session, without the attacker's interference, packets that match loose rules only occurs when the session is in a specific state. So loose rules still exist in many Snort rule sets using *flowbits*. Moreover, loose rules are easier to write and reckless users normally create loose rules in their policy. Because loose rules are easy to be triggered and normally cause false positives, they might be exploited to

cause a difference between the actual and in-snort session states when *flowbits* options are used to keep track of the session state. Each rule when triggered is supposed to change the in-snort session state according to the actual connection state. A loose rule might be exploited by the attacker to change the in-snort session state but preserve the actual session state.

3.3.3 *Flowbits* Misconfiguration Problem

Problem definition: Let $S = \{R_1, R_2, \ldots, R_n\}$ be a Snort rule set which uses *flowbits*. Let T be a subset of S ($T \subset S$) that raise alerts. We denote T as the set of all target rules of S. The problem is to find all possible packet sequences that successfully attack the service protected by S yet manage to not trigger any rule from T.

A target rule is normally a non-intermediate rule (usually with high priority) which indicates a successful attack when it is triggered. When *flowbits* is used in a rule set, many rules act as intermediate steps leading to the triggering of a target rule if the attack is successful. Intermediate rules normally don't raise alerts and this can be done by using "noalert" with *flowbits*. *flowbit:noalert* is a critical function. It enables us to use a rule that would hit on a lot of traffic that is not of interest, but that must occur before a packet that would be of interest in a session [10].

Definition 5 (Target rule set) *It is the set of all target rules in a rule set.*

Definition 6 (Target rule group) *It is a group of target rules that have the same match options except the flowbits conditions.*

It is conceptually possible that a rule set contains several target rule groups. In this case, we need to define and solve the problem for each target rule group separately so that to detect evasions that try to mislead Snort to trigger an alarm different than the one the attack really does.

A packet sequence can include packets coming from both directions: server to client and client to server. From the attacker's perspective, a packet can come from the attacker's side and the other side. However, in most of the time, the attacker plays the client role. So in this thesis, we assume that the attacker always comes from the client side, and he tries to carry out an attack to a service at a server and does not want to be detected by Snort. Nevertheless, the attack concept is still applicable if the attacker is from the server side.

In this thesis, we only consider the session states considered by Snort because Snort might not need to use all possible states from a real session to detect a specific attack. It is very important to understand two crucial concepts: the *actual session state* and the *in-snort session state*. The actual session state is the state where the session is truly currently in, and the in-snort session state is the state that Snort thinks the session is currently in.

Definition 7 (Session state) *It is defined as a group of labels that are currently set. If* n *is the number of labels used in the rule set, then there are potentially* 2^n *different session states.*

Definition 8 (Target state) *It is a session state where a target rule in T can be triggered (to change to another state).*

Definition 9 (Non-target state) *It is a session state where no target rule in T can be triggered (to change to another state).*

Definition 10 (Target packet) *It is a packet that matches any target rule in T and presumably the last packet in the packet sequence of a real attack.*

Definition 11 (Evadable rule) *It is a rule that can be triggered by the attacker to change the in-snort session state but preserve the actual session state. A rule is evadable or not depends on whether the attacker can construct a packet that can trigger the rule without affecting the actual session. This is discussed in Section 3.3.5.*

An evadable rule can be triggered by two different kinds of packets: a packet (from the connection session) that is supposed to trigger the rule at a given time and correctly reflects what Snort thinks about the session, and a packet that is not supposed to trigger the rule and makes Snort misjudge the session. Given a rule R_i, let P_i represent the first kind of packets, called "a normal packet", and P_i* represent the second kind of packets (if R_i is evadable), called "an evasion packet". P_i* causes a change in the in-snort session state (by triggering R_i), but has no effect on the actual session state.

A packet sequence is considered a successful attack if and only if it puts the session in one of the target states right before the corresponding target packet (the last packet in the sequence) is sent. Therefore, a packet sequence is considered a successful attack but does not trigger the target rule if right before the target packet is sent; the packet sequence puts the actual session in one of the target states and puts the in-snort session in one of the non-target states.

We can assume that, when the actual session is in one of the target states and the in-snort session is in one of the non-target states, the attacker will always trigger the sending of the corresponding target

packet. As a result, the problem can be redefined as finding all possible packet sequences that put the actual session in one of the target states and the in-snort session in one of the non-target states. Moreover, a rule set can detect all packet sequences that trigger a target rule in T if and only if it can detect all packet sequences that put Snort in a target state. So from now on, we don't consider a packet sequence that includes a target packet as its last packet because the target packet can be implicitly added to the end of the packet sequence when the attacker performs the real attack.

3.3.4 Language of All *Flowbits* Evasion

From the rule set S, a corresponding state diagram can be created to show all possible reachable session states and transitions between them. This state diagram can be considered as a DFA. Because we deal with two separate session states: the actual one and the in-snort one, each of these session states corresponds to a DFA, say D_s and D_a. Both D_s and D_a have the same alphabet, set of states, start state and the set of accept states. The only difference between them is the transition function. They are constructed as shown in Table 3.2.

1: *Set of states*: reachable session states constructed from the rule set
2: *Start state:* the state where no label is set
3: *Accept state:* all target states
4: *Alphabet* $\sum$ = {P_i: R_i is not a target rule} ∪ {P_i*: R_i is evadable and R_i is not a target rule}
5: *//Transition function*
6: **for all** non target rule R_i **do**
7: **for all** state A **do**
8: **if** R_i can be triggered at A leading to state B **then**
9: Add (A,P_i) → B to both D_s and D_a
10: **if** R_i is evadable **then**
11: Add (A,P_i*) → B to D_s
12: Add (A,P_i*) → A to D_a
13: **end if**
14: **end if**
15: **end for**
16: **end for**

Table 3.2: Construction of in-snort (D_s) and actual (D_a) session DFAs

Let $L_s(S)$ and $L_a(S)$ be the languages corresponding to D_s and D_a respectively. While $L_s(S)$ represents all packet sequences that Snort thinks to put the session in a target state, $L_a(S)$ represents all possible packet sequences that truly put the session in a target state.

The goal is then to find all possible packet sequences that truly put D_a in a target state but not D_s. These packet sequences must hence be accepted by D_a and rejected by D_s. In other words, these packet sequences are accepted by both the D_a and $\neg D_s$. If we consider these packet sequences as a language, say $L_e(S)$, then:

Language of all *flowbits* evasion: *The language of all packet sequences (or evasion sequences) that successfully attack the service protected by a rule set S without triggering an alarm is equal to:*

$L_e(S) = L_a(S) \cap \neg L_s(S) = L(D_e(S))$ where $D_e(S) = D_a(S) \cap \neg D_s(S)$.

$L_e(S)$ therefore represents all possible packet sequences (or ***evasion sequences***) that successfully attack the service without the Snort's detection. If $L_e(S)$!= $\varnothing$, S is said vulnerable. Table 3.3 shows steps to find all possible evasion sequences.

1: Construct $D_s(S)$ and $D_a(S)$ based on the rule set S and evadable rules in S
2: Construct $\neg D_s(S)$
3: Construct $D_e(S) = D_a(S) \cap \neg D_s(S)$
4: $L_e(S) = L(D_e(S))$

Table 3.3: Steps to find all possible evasion sequences

3.3.5 Rule evadability

An important task the attacker needs to do to exploit the *flowbits* misconfiguration is to construct a packet which causes the triggering of a given rule to change the in-snort session state but does not have any effect on the actual session state. In practice, there are many possible conditions that can be exploited to construct such a packet. For example, if the Snort IDS is not correctly configured (e.g: some preprocessors are not turned on), existing NIDS evasion techniques [44] [20] [32] can be used to accomplish this task. However, in this thesis, we concentrate on exploiting two existing factors to create evasion packets: the packet-based property of Snort and loose rules.

Besides, we also discuss the possibility of constructing evasion packets for two kinds of rules: rules triggered by packets coming from the client side and rules triggered by packets coming from the server side. It is always easier for the attacker to control packets coming from the client side (the attacker's side) than the server side.

3.3.5.1 Packet-based Property of Snort

As mentioned above, Snort is a packet-based NIDS and most packets received by Snort are checked by the detection engine. This feature allows the creation of evasion packets. There might be many ways to achieve this, but one method is to construct a packet that matches a given rule, however, with "out of order" TCP sequence number. This packet is not processed by the receiver's application layer but still examined by the Snort detection engine and then triggers the rule. For example, if the expected sequence number of the next packet (from the client) is *X*, we can construct a packet with sequence number *X-1*. Snort with *stream5* preprocessor enabled knows that the packet does not have an expected sequence number for the session stream and is overlapped with a previous packet (assuming that this previous packet has some payload). Snort does exactly as the protected host does: not reassemble this packet into the session stream. However, the packet is still passed down to the detection engine because the packet might match TCP-based attacks where sequence number is not important (e.g: Nmap [31] uses TCP packets with random sequence number to probe a host's OS). This issue was tested with Snort 2.8.1 [41] (the newest version at this time).

Packets constructed by exploiting this feature are injected into the connection session with the purpose of faking interactions (requests and responses) between the client and the server to make Snort misunderstand the session. There are two cases according to two kinds of rules:

Rules matching traffic from the client side: It is always possible to construct a packet (with "out-of-order" sequence number) to fake a request from the client and inject it into the connection session. Therefore, any rule matching traffic from the client side can be triggered without causing the actual session state change.

Rules matching traffic from the server side: It is easy to fake a response from the server but it is hard to deliver the packet to the Snort IDS. In order to deliver a packet to the Snort IDS, the attacker needs to have control over a computer from a network where the crafted packet can reach the Snort IDS (e.g.: a computer from the same network with the server the attacker is trying to attack). Notice that even though the attacker is in the same network with the server, carrying the attack directly from local is still detected if the Snort IDS is configured to detect attacks carried out in the local network. Therefore, constructing fake responses from the server is still necessary.

The scenario where the attacker can control a computer from which the packet can be delivered to the Snort IDS is not rare. For example, an employee from a company has his own computer or a student has an account to access a server in the university network, or the attacker has compromised a

computer in the network before, etc. In this case, the Snort IDS thinks that the packet comes from the server and processes it normally.

In addition, when dealing with rules matching packets from the server side, sometimes the injected packet needs to be repeatedly sent to make sure that it comes to the Snort IDS before the real response from the server. When the real response reaches the Snort IDS, the in-snort session has already been in a state where the real response is no longer important, however, the injected response is ignored by the client and the real response is processed normally at the client side. This situation occurs when there is more than one possible response from the server for a given request, and based on the response from the server, the in-snort session state is changed accordingly.

3.3.5.2 Loose Rules

When a rule is loose, it does not thoroughly match a packet by using tight options like *dsize*, *depth*, *offset*, etc. A loose rule can wrongly explain the intention of a packet if the packet just happens to match the rule but logically does something else. Moreover, it is very possible to create or trigger the sending of such a packet. The packet can be created from the connection session itself (no need to be crafted and injected). There are 2 cases according to two kinds of rules:

Rules matching traffic from client: Although depending on the service protocol, most of the time the attacker can easily make a request from client that matches a loose rule but logically does something else rather than the rule expected. For example, a loose rule checks if a user currently in a FTP session is trying to quit the session by examining packets from the client to see if any packet has "QUIT/n" in its payload. The attacker can make a request to create a directory named "QUIT", which happens to have ""QUIT/n" in the packet payload and causes Snort misjudge the session.

Rules matching traffic from server: It is harder to evasively trigger a loose rule matching traffic from the server side than the client side because traffic from the server side is not always controllable. However, when dealing with interactive protocols, there are many tricks the attacker can use to cause the server to send a packet containing desired strings and then trigger the rule. Let's say if a rule simply checks for any packet from the server in a telnet session which contains the string "Granted" in the payload, the attacker can issue an invalid command containing "Granted". The server will send back a complain of unknown command, which happens to contain the command name "Granted", and hence triggers the rule. Another trick is the attacker can create a folder with name "Granted" and then try to list all the folders.

3.3.5.3 Summary

In summary, a rule's content and the scenario context decide whether a packet can be constructed to trigger the rule without affecting the actual session state. In other words, whether a rule is evadable or not depends on its content and the scenario context.

If a rule matches traffic from client: it is always evadable because the attacker can exploit the packet-based property of Snort to construct a corresponding evasion packet. If the rule is also loose, the attacker has another choice (exploiting the rule looseness) to accomplish the task. On the other hand, if a rule matches traffic from server: the probability the rule is evadable depends on several aspects: whether the attacker has access to a computer where fake responses (from the server) can reach the Snort IDS, service protocol, and the rule's looseness. If the rule is loose, it is very likely that the rule is evadable. Otherwise, the rule is evadable with small possibility. Table 3.4 shows the algorithm to determine the evadability of a rule.

Algorithm FindRuleEvadability
Input: Rule *R*
Output: *the evadability of rule R*
1: **if** R matches traffic from client **then**
2: **return** *EVADABLE*
3: **end if**
4: **if** R contains only *content* options **then**
5: **return** *PROBABLY_EVADABLE*
6: **else**
7: **return** *MAYBE_EVADABLE*
8: **end if**

Table 3.4: Algorithm to determine the evadability of a rule

3.3.6 Example

The rule set in Table 3.5 is created to follow an FTP session. It will raise an alert if a normal user tries to do anything related to an important file which should only be accessed by the Admin. Rule 1 and Rule 2 determine if a normal user is logging in, Rule 3 indicates that the user is denied to login, Rule 4 checks if the user is successfully logged in, Rule 5 indicates that the user has logged out of the FTP session, and Rule 6 checks if the logged-in user tries to do anything with the file C:\Windows\System32\sam and raises an alert. Only rule 6 is the target rule.

R1	alert tcp 192.168.0.101 any -> 192.168.0.100 21 (msg:"FTP Normal User Login Attempt - Send username"; flow:established, to_server; content:"USER"; depth:5; nocase; content:!"Admin"; within:5; content:"|0D0A|"; *flowbits:set,NormalUserLoginAttempt; flowbits:noalert;*)
R2	alert tcp 192.168.0.101 any -> 192.168.0.100 21 (msg:"FTP Normal User Login Attempt - Send password"; flow:established, to_server; content:"PASS"; depth:5; *flowbits:isset, NormalUserLoginAttempt; flowbits:set,NormalUserLoginAttempt2;flowbits:noalert;*)
R3	alert tcp 192.168.0.100 21 -> 192.168.0.101 any (msg:"FTP login denied"; flow:established,to_client; flags:A**;** *flowbits:isnotset, NormalUserLoggedIn; flowbits:isset,NormalUserLoginAttempt; flowbits:isset,NormalUserLoginAttempt2; flowbits:unset,NormalUserLoginAttempt; flowbits:unset,Normal UserLoginAttempt2; flowbits:noalert;*)
R4	alert tcp 192.168.0.100 21 -> 192.168.0.101 any (msg:"FTP login granted"; flow:established,to_client; content:"230 Login successful.|0D0A|"; nocase; flags:AP;*flowbits:isset,NormalUserLoginAttempt2; flowbits:set,NormalUserLoggedIn;flowbits:noalert;*)
R5	alert tcp 192.168.0.101 any -> 192.168.0.100 21 (msg:"FTP user exits"; flow:established,to_server; content:"QUIT|0D0A|"; nocase; *flowbits: isset,NormalUserLoggedIn; flowbits:unset, NormalUserLoggedIn; flowbits:unset,NormalUserLoginAttempt; flowbits:unset, NormalUserLoginAttempt2; flowbits:noalert;*)
R6	**alert tcp 192.168.0.100 any -> 192.168.0.101 21 (msg:"Normal User accesses important file"; flow:established, to_client; content:"Windows";content:"system32";content:"sam"; nocase;*flowbits:isset, NormalUserLoggedIn;*)**

Table 3.5: A rule set to detect a non-admin user accessing an important file from a FTP session

Theoretically, any rule from the rule set is evadable, but in this example, for simplicity, we assume that only Rule 5 is evadable from the attacker's perspective. It means that the attacker can come up with a way to trigger Rule 5 without affecting the actual session state. The attacker can accomplish this in many different ways which are discussed in Section 3.3.5.

There are 3 labels used in this rule set which put the session in $2^3 = 8$ possible states. However, there are only 4 reachable states including the empty state (no label is set): A ={}, B={NormalUserLoginAttempt}, C={NormalUserLoginAttempt, NormalUserLoginAttempt2} and D={NormalUserLoginAttempt, NormalUserLoginAttempt2,NormalUserLogedIn}

The in-snort DFA D_s and actual session state DFA D_a are depicted in Figure 3.1 and Figure 3.2. $\neg D_s$ is constructed as in Figure 3.3. The intersection of $\neg D_s$ and D_a, which is D_e is constructed in Figure 3.4, where (A,A) is the start state and (D,A), (D,B), and (D,C) are accept states. The language corresponding to this D_e represents all possible packet sequences that successfully attack the FTP server.

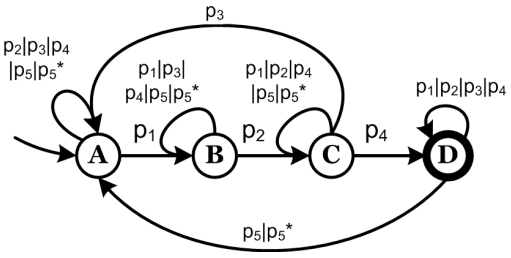


Figure 3.1: D_s of the FTP rule set

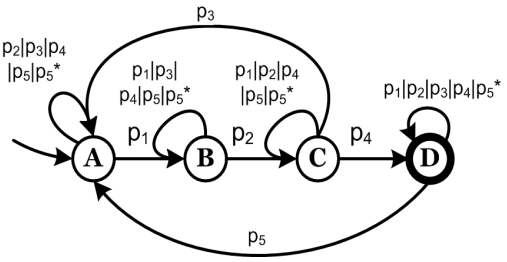


Figure 3.2: D_a of the FTP rule set

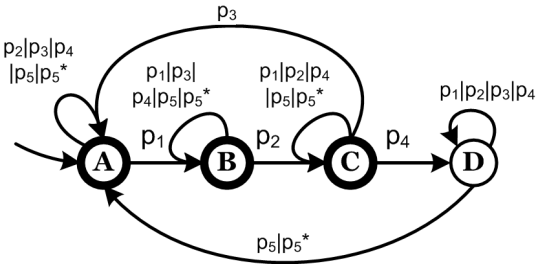


Figure 3.3: $\neg D_s$ of the FTP rule set

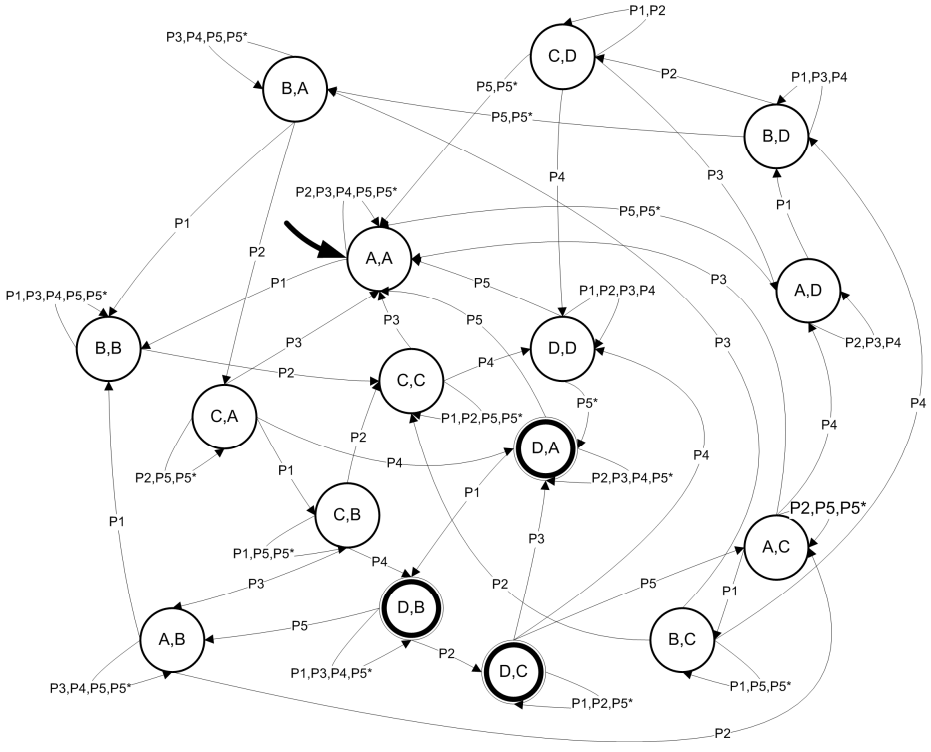


Figure 3.4: D_e of the FTP rule set

For example, P_1 P_2 P_4 P_5* is a packet sequence accepted by this D_e. The attacker can apply this packet sequence to perform a real attack. The attack is carried out as follow: The attacker logs in as a normal user (not Admin) with correct username and password. This action needs P_1 and P_2 to be sent from the attacker and leads to the sending of P_4 from the server to indicate that the user is successfully authorized. The next step the attacker needs to do is to cause the sending of P_5*. There are 2 options to create P_5^*. The first option is to manually construct and inject into the connection a packet that matches Rule 5 but has out-of-order sequence number. The second option is to send a packet that matches Rule 5 but logically does something else rather than exiting the session as Snort thinks. The attacker can create a directory named "QUIT", which makes Snort misjudge the session and think that the user has logged out. After that, the attacker can take the last step of the attack which is downloading or accessing the restricted important file at the server. This action won't trigger the target rule.

3.3.7 *Flowbits* Misconfiguration Rectification

3.3.7.1 Manual Solution

The admin needs to re-examine rules that he thinks are evadable. If the admin is not sure which rule is really evadable, it is safe to assume that all rules are evadable and constructs all possible evasion sequences as shown previously to see which rules can be exploited to evade Snort if they are evadable. These rules should be rewritten in a tighter manner: matching different packet fields (from different layers like TCP, IP) by using non-payload options; the packet payload should be carefully matched by using tight options like *offset, depth*, etc. Moreover, if possible, the admin should limit rules matching packets coming from the attacker's side as much as possible because rules that match packets coming from the protected side (normally from server) are harder to evade, while rules matching packets coming from the client side are always evadable as discussed in Section 4.

3.3.7.2 Ideal Solution

An ideal solution is the solution that automatically changes the rules so that the new rule set semantically does the same thing as the old rule set but is not vulnerable to the proposed evasion technique. However, the difficulty to find such an ideal solution is to automatically understand the

semantics of the rule set. Moreover, even if the rule set's semantics is understood, it is possible that any rule set which follows exactly the semantics is always vulnerable to the proposed attack. Therefore, finding an ideal solution is very hard and sometimes impossible.

3.3.7.3 Proposed Solutions

In this section, we suggest solutions to patch vulnerable misconfigured *flowbits* rule sets. The idea is to create a new rule set that detects not only all attack sequences but also all evasion sequences. If the number of evasion sequences is small, extra rules can be added to detect each of these sequences without causing much overhead (number of added rules). Normally, a small vulnerable rule set has a small number of evasion sequences. Besides, there might be an exponential number of evasion sequences, so patching each evasion sequence works effectively for only small rule sets. On the other hand, if the rule set is large (the number of evasion sequences is also large), another approach is used to patch the rule set based on the common features among all evasion sequences: always contain at least one evasion packet. Moreover, a target packet always follows an evasion sequence in a real attack is an aspect that can be considered as well.

A. Solution for small rule sets

Our proposed solution for small rule sets is trying to detect all possible evasion sequences by adding extra rules to the old rule set with acceptable overhead. We can simultaneously consider the D_e as a directed graph, where each state is a node and each transition is a directed link. Theoretically, we need to add a rule set to detect each path from the start state to an accept state (called an evasion path), however, the number of evasion paths is possibly infinite and we cannot add infinite number of rules to Snort. We need to find a way to add minimum number of rules but still possibly detect all evasion paths (proved in Appendix A.2). It turns out that it is enough to consider only simple evasion paths (a simple path is a path that does not have a circle) because rules added to detect all simple evasion paths can detect all evasion paths. Moreover, it is sufficient to consider only subset paths over all simple paths. Here is the procedure to create new rules to add to the old rule set:

1) Determine all simple paths from the start node to an accept node from the graph, let *SP* be the set of these simple paths.

2) If $x,y \in SP$, x's corresponding packet sequence is a prefix of y's corresponding packet sequence, remove y from *SP*.

3) Assign each simple path in *SP* a different number.

4) For the simple path number *k* in SP, a rule set is created as follow:

a) A label is created for each node, assume the simple path has *n* nodes, so they are labeled "A-k1", "A-k2",… "A-k*n*" respectively as the order of the nodes on the path (so each simple path uses a different label set).

b) For each directed link on the path: create a new rule. Assume that the link connects a node with label "A-k*i*" to a node with label "A-k(*i+1)*", and the link's attribute is P_j or P_j*. A new rule $R_k(i)$ is created by using all options in R_j in the original rule set (header and body) except the *flowbits* options. $R_k(i)$'s *flowbits* options check if label "A-k*i*" is set, and also set label "A-k(*i+1)*". "*noalert*" option is also used in the rule because rules created for the this simple path are not target rules. Note: "A-k1" is always set by default, so $R_k(1)$'s *flowbits* options only need to set label "A-k2".

c) For the accept node (the last node on the path having label "A-k*n*"), create the rule $R_k(n)$ by using all options in an inspected target rule (header and body) except the *flowbits* options. $R_k(n)$'s *flowbits* option checks if label "A-k*n*" is set. $R_k(n)$ acts as the target rule in this rule set.

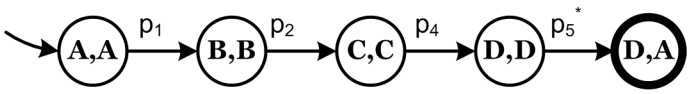


Figure 3.5: Simple evasion path(s) collected from D_e

The set of simple paths *SP* collected from the D_e of Figure 3.4 has one simple path (after removing subset paths) as shown in Figure 3.5. There are five rules added for this simple path as shown in Table 3.6.

The fact that the procedure searches for all simple paths that reach target states makes its running time potentially prohibitive. We prove in Appendix C.2 that the complexity of this phase ranges from M^2 (D_e consisting of a single simple path) to $|T| \times (M^2 - |T| - 1) \times |\Sigma|^{M^2 - |T| - 1}$, where M is the number of states of D_s, |T| is the size of the target rules set and |Σ| is the size of the rule set plus the number of evadable rules.

Because of this complexity, this solution is only suitable for rule sets of a small size. In the following, we present a different solution that is feasible for large rule sets.

R_1(1)	alert tcp 192.168.0.101 any -> 192.168.0.100 21 (msg:"FTP Normal User Login Attempt – Send username"; flow:established,to_server; content:"USER"; depth:5;nocase; content:!"Admin"; within:5; content: "|0D0A|";*flowbits*:set,**A-12**;*flowbits*:noalert;)
R_1(2)	alert tcp 192.168.0.101 any -> 192.168.0.100 21 (msg:"FTP Normal User Login Attempt - Send password"; flow:established, to_server; content:"PASS"; depth:5; *flowbits*:isset, **A-12**; *flowbits*:set, **A-13**; *flowbits*:noalert;)
R_1(3)	alert tcp 192.168.0.100 21 -> 192.168.0.101 any (msg:"FTP login granted"; flow:established,to_client; content:"230 Login Successful.|0D 0A|"; nocase;flags:AP;*flowbits*:isset,**A-13**;*flowbits*:set,**A-14**;*flowbits*:noalert;)
R_1(4)	alert tcp 192.168.0.101 any -> 192.168.0.100 21 (msg:"FTP user exits"; flow:established,to_server; content:"QUIT|0D0A|"; nocase; *flowbits*:isset,**A-14**;*flowbits*:set,**A-15**;*flowbits*:noalert;)
R_1(5)	alert tcp 192.168.0.100 any -> 192.168.0.100 21 (msg:"Normal User accesses important file"; flow:established,to_client; content:"Windows"; content:"system32";content:"sam"; nocase; *flowbits*:isset,**A-15**;)

Table 3.6: Rules added for the simple path in Figure 3.5

B. Solution for large rule sets

We want to detect all evasion sequences but sometime we cannot find all of them or maybe there are too many of them to deal with, which cause too much overhead. We need a way to cover all evasion sequences without considering each of them separately. We know that any evasion sequence needs to exploit at least one evadable rule, i.e. it contains at least one evasion packet. Furthermore, in a real attack, a target packet is always sent after an evasion sequence.

The idea is to set a flag whenever an evadable rule is triggered. After that, if Snort sees the target packet and the flag is set, it will raise an alert. However, we do not need to do this with all evadable rules. An evadable rule needs to set the flag when it is triggered only if the rule causes the rule set vulnerable ($L_e(S) != \varnothing$) even if all the other rules are not evadable. This kind of evadable rule is said vulnerable. The reason is if two or more evadable rules together cause the rule set vulnerable, then at least one of them is vulnerable (this is proved in Appendix C.1). Therefore, an evasion sequence always contains an evasion packet whose corresponding evadable rule is vulnerable.

Here are the steps to patch a large vulnerable rule set:

1) Check if any evadable rule is vulnerable. This can be done by creating the D_e for each evadable rule (while assuming all the other rules are not evadable) and check if $L_e(S)$!= ∅ (or D_e has a reachable accept state).

2) For each vulnerable rule R_i, insert the *flowbits* option *flowbits:set, TARGET_RULE_**X*** where ***X*** is a representative number for the target rule group currently inspected.

3) Create a new rule that uses all options in an inspected target rule except the *flowbits* options. This rule's *flowbits* option checks if the label *TARGET_RULE_**X*** is set.

These steps have a polynomial time complexity. In Appendix C.3, we prove that its worse case running time is |R| x (M^2 x |Σ| + 1), where |R| is the size of the rule set, M is the number of states of D_s, and |Σ| is the size of the rule set plus the number of evadable rules.

Example: Instead of giving an example of a large rule set, we use the rule set in Table 3.5 for simplicity because the approach works for small rule sets as well. Assume that all rules in Table 3.5 are evadable. The first step indicates that only rule R_3 and R_5 are vulnerable (note that Figure 3.4 is the D_e created for R_5). So R_3 and R_5 are modified by inserting the *flowbits* option *flowbits:set, TARGET_RULE_6*. R_7 is the added rule and all other rules are the same. Table 6 shows the modified and added rules to patch the rule set.

R_3	alert tcp 192.168.0.100 21 -> 192.168.0.101 any (msg:"FTP login denied"; flow:established,to_client; flags:A**;** *flowbits*:isnotset, NormalUserLoggedIn; *flowbits*:isset,NormalUserLoginAttempt; *flowbits*:isset,NormalUserLoginAttempt2; *flowbits*:unset,NormalUserLoginAttempt; *flowbits*:unset, Normal UserLoginAttempt2; ***flowbits*: set, TARGET_RULE_6**; *flowbits*: noalert;)
R_5	alert tcp 192.168.0.101 any -> 192.168.0.100 21 (msg:"FTP user exits"; flow:established,to_server; content:"QUIT|0D0A|"; nocase; *flowbits*: isset,NormalUserLoggedIn; *flowbits*:unset, NormalUserLoggedIn; *flowbits*: unset,NormalUserLoginAttempt; *flowbits*:unset, NormalUserLoginAttempt2; ***flowbits*: set, TARGET_RULE_6**; *flowbits*: noalert;)
R_7	alert tcp 192.168.0.100 any -> 192.168.0.101 21 (msg:"Normal User accesses important file"; flow:established, to_client; content:"Windows";content:"system32";content:"sam"; nocase; ***flowbits*: isset, TARGET_RULE_6;**)

Table 3.7: Modified and added rules using the large rule sets approach

C. False positives discussion

Even though the approach used for large rule sets also works for small rule sets, it potentially causes more false positives then the one used for small rule sets. While the latter only raises an alert if a complete evasion sequence is seen, the former raises an alert on important packets in an evasion sequence. However, the overhead caused by the latter is larger than the former. Therefore, depending on the size of a rule set and the acceptable overhead, an appropriate solution can be applied.

A patching solution without completely changing the rule set (which requires the understanding of the rule set semantics) always potentially causes false positives. In the extreme case, when all rules are vulnerable, the only solution is to always raise an alert when an evasion packet and a target packet are seen. This reflects the situation when Snort cannot trust any packet and it has to raise an alert most of the time in the favor of not missing the attack.

However, our proposed solutions should not cause false positives in general because the patched portion covers possible events seen from the attacker and normally not occurring from normal users. For example, in an FTP session, a normal user will not access a file after he quits the session. Moreover, in practice, the extreme case does not seem to exist, as shown in the evaluation section of this chapter.

D. False positives control patch

Although our solutions are only potential to cause false positives, there are situations (like the extreme case mentioned above) where false positives are very likely to occur. In this section, we propose a false positives control patch in addition to the proposed solutions. False positives occur when alerts are raised on normal users' actions. Therefore, in order to avoid false positives, Snort needs to consider session packets it will see after it is put into a target state (by a patched rule set: a vulnerable rule set rectified by the small rule set solution or the large rule set solution). These packets might give Snort some clues about who is running the session. If Snort sees a target packet right away, it is most likely that the attacker is running the session. Otherwise, if the following packets cause transitions in D_s as a normal user will do, it is possible that the session is run by a normal user. The more session packets Snort considers afterward, the more accurate decision Snort will make.

Solutions for small and large rule sets put Snort into a target state when an evasion sequence is seen or a vulnerable rule is triggered respectively. Now, instead of raising an alert whenever a target packet is seen after Snort is put into a target state, if a number of packets corresponding to normal

users' actions are seen (before a target packet), Snort will assume that the session is run by a normal user and put the session back to a non-target state.

In order to determine all possible actions a normal user might do after Snort is put into a target state, we need to know all the states in D_s after an evasion sequence is seen (for small rule sets solution) or after a vulnerable rule is triggered (for large rule sets solutions). Then all possible actions of a normal user are equivalent to all paths starting from any of these states.

Let L be the length of actions (or length of a path) Snort considers afterward.

Here is the procedure to add new rules (to the patched rule set from proposed solutions) to control potential false positives:

A. Find all states in D_s that are used to determine possible normal users' actions:

1) Solution for small rule sets: each evasion sequence puts D_s into a state and this state can be found by tracing through D_s. States are collected for all evasion sequences.

2) Solution for large rule sets: for each vulnerable rule, find all states in D_s where the rule can be triggered (to change to a different state), and then find all states right after the vulnerable rule is triggered. States are collected for all vulnerable rules.

B. For each state X_t found at stage A:

1) Find all paths from D_s of length L started at state X_t

2) Assign each path a different number.

3) For the path number k:

a) A label is created for each node. The first node is labeled as the last node of the simple path (from D_e) corresponding to the evasion sequence that puts D_s into X_t (for small rule sets solution) or *TARGET_RULE_**X*** (for large rule sets solution). The rest are labeled "A-t-k2", "A-t-k3",…,"A-t-kn" respectively as the order of the nodes on the path.

b) For the first link on the path: assume the link's attribute is P_j (or P_j^*) . Create a rule $R^t(1)$ uses all options in R_j in the original rule set (header and body) except the *flowbits* options. $R^t(1)$'s *flowbits* options check if the first node's label is set, and then set label "A-t-k2". "noalert" option is used in this rule.

c) For the last link on the path: assume the link's attribute is P_j (or P_j^*) . Create a rule $R^t(n-1)$ that uses all options in R_j in the original rule set (header and body) except the *flowbits* options. $R^t(n-1)$'s *flowbits* options check if label "A-t-k(n-1)" is set, and then unset the first node's label. "noalert" option is used in this rule.

d) For each link on the path (except the first and the last links): assume that the link connects a node with label "A-t-k*i*" to a node with label "A-t-k(*i+1*)", and the link's attribute is P_j (or P_j^*). A new rule $R^t(i)$ is created by using all options in R_j in the original rule set (header and body) except the *flowbits* options. $R^t(i)$'s *flowbits* options check if label "A-t-k*i*" is set, and then set label "A-t-k(*i+1*)". "noalert" option is used in the rule.

L can be chosen by Snort users to balance between false positives and overheads (number of rules added). It is easy to see that the longer L, the less false positives, but the more rules to be added.

There is always a tradeoff between vulnerability and false positives. In the extreme case where Snort needs to raise an alarm most of the time, the new rule set is totally secure. On the other hand, if the rule set is not patched and insecure, it will not cause any new false positive. This false positives control patch makes the rule set vulnerable again because a smart attacker can always send packets corresponding to all possible actions a normal user might do before sending the target packet. However, this patch at least gives Snort users an option to control potential false positives and is useful when missing some attacks is better than having too many false positives. Moreover, Snort users can apply the heuristic method to reduce number of rules added to control false positives. Instead of adding rules to cover all paths found in step 1 (stage B) above, only paths that reflect regular normal users' actions should be considered (these actions can be found from the application session log files).

$R^1(1)$	alert tcp 192.168.0.101 any -> 192.168.0.100 21 (msg:"FTP Normal User Login Attempt – Send username"; flow:established,to_server; content:"USER"; depth:5;nocase; content:!"Admin"; within:5; content: "|0D0A|"; ***flowbits*:isset,A-15; *flowbits*:set,A-1-12; *flowbits*:noalert;**)
$R^1(2)$	alert tcp 192.168.0.101 any -> 192.168.0.100 21 (msg:"FTP Normal User Login Attempt - Send password"; flow:established, to_server; content:"PASS"; depth:5; ***flowbits*:isset, A-1-12; *flowbits*:unset, A-15; *flowbits*:noalert;**)

Table 3.8: False-Positives-Control Rules added to the solution in Table 3.6 when L=2

Example: There are two rules (Table 3.8) added to the solution in Table 3.6 (solution for small rule sets) to control false positives where L=2.

3.3.8 Proof of Correctness

There are two things that we need to prove for each solution approach:

1) The new rule set is complete: it can still detect all packet sequences detected by the old rule set and also detect all possible evasion sequences.

2) The new rule set is sound: The new rule set itself is not vulnerable to the proposed evasion technique.

3.3.8.1 Solution for small rule sets

In this section, we introduce a crucial concept: *independent rule sets.* Two rule sets are independent if the triggering of any rule in one rule set does not depend on the existence of the other rule set. This concept is important to argue the soundness and completeness of the new rule set.

We can assume that rules using *flowbits* are all *alert* rules. Moreover, another assumption we can make is: the option *flowbits:reset* is not used in a rule set. This option *flowbits:reset* is used without specifying any label and equivalent to unsetting all labels already set for a flow. This option can be replaced by using *flowbits:unset* on all labels of a rule set.

With these assumptions, two rule sets which use the *flowbits* option on two non-overlapping label sets are independent. Assume S_1, S_2 ... ,S_n are added rule sets for n simple paths in *SP.* We have S_{new} = $S \cup S_1 \cup S_2 \ldots \cup S_n$.

- ***Completeness of the new rule set***

In order to prove that the new rule set can detect all packet sequences detected by the old rule set, we need to prove that $L_s(S) \subset L_s(S_{new})$ and this is proved in Appendix A.1 Moreover, we need to prove that the new rule set also detect all possible evasion sequences. In other words, we need to prove that, given any evasion sequence X_{es} accepted by $D_e(S)$, the new rule set can detect X_{es}. i.e., $X_{es} \in L_s(S_{new})$. This is proved in Appendix A.2.

- ***Soundness of the new rule set***

In order to show that the new rule set S_{new} is safe against the proposed attack, we need to prove $L_e(S_{new}) = \varnothing$. In order to prove this, we first need to prove that if $S = S_1 \cup S_2$ (S_1 and S_2 are independent), then $L_s(S) = L_s(S_1) \cup L_s(S_2)$ and $L_a(S) = L_a(S_1) \cup L_a(S_2)$.

Then we prove that each added rule set corresponding to a simple path is not vulnerable to the proposed attack. Finally, by using induction, we can prove $L_e(S_{new}) = \varnothing$. The proof can be found in Appendix B.

3.3.8.2 Solution for large rule sets

Let S_{new} be the new constructed rule set. We'll prove that S_{new} is complete and sound.

- ***Completeness of the new rule set***

The way the new rule set is constructed can be viewed from the DFA creation. It is equivalent to adding extra transitions and states to the $D_s(S)$: for any state X, for any vulnerable rule R_i, if (X,P_i^*) != X then add a state X* and transitions $(X,P_i^*) = X^*$, $(X,P_i) = X^*$. Set X* as a target state. Now we have an NFA_s of the new rule set. It is easy to see that any sequence accepted by $D_s(S)$ is accepted by this NFA_s, i.e, $L_s(S) \subset L_s(S_{new})$. Figure 3.6 shows an example of the NFA_s of the new rule set S_{new} for the vulnerable rule set in Table 2 with the assumption that only rule 5 is evadable (and the rule is vulnerable).

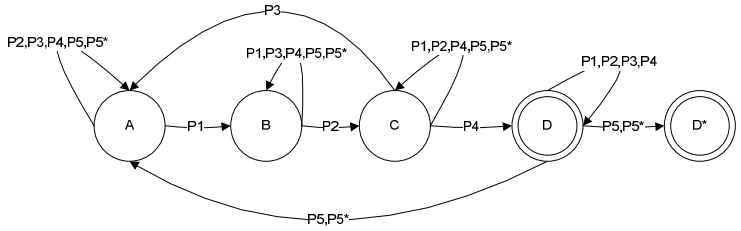


Figure 3.6: NFA_s of the new rule set S_{new}

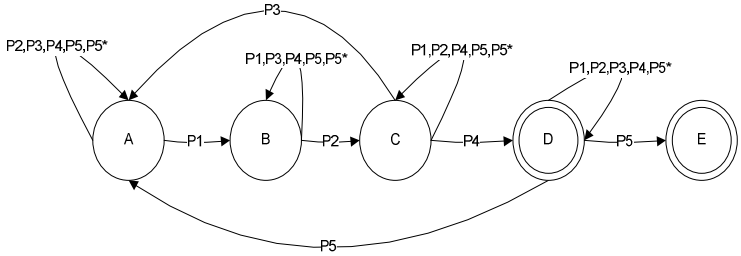


Figure 3.7: NFA_a of the new rule set S_{new}

To prove that all evasion sequences can be detected by the new rule set, we first need to prove a theorem that if two or more evadable rules together cause the rule set vulnerable, then at least one of them is vulnerable. This theorem is proved in Appendix C.1. Therefore, any evasion sequence needs to trigger at least one vulnerable rule, otherwise, there exists an evasion sequence that does not contain any evasion packet whose corresponding rule is vulnerable, and this contradicts the theorem. From the way the new rule set is constructed, after a vulnerable rule is triggered, Snort is always put in a target state, hence given any evasion X_{es}, we have $X_{es} \in L_s(S_{new})$. So, all evasion sequences can be detected by the new rule set.

- ***Soundness of the new rule set***

We can convert $NFA_s(S_{new})$ to $NFA_a(S_{new})$ by replacing transitions $(X,P_i^*) \rightarrow Y$ by $(X,P_i^*) \rightarrow X$ for all vulnerable rules R_i's. Figure 3.7 shows the corresponding NFA_a of the new rule set S_{new} in Table 4.

We see that any sequence accepted by $D_a(S)$ is accepted by $NFA_a(S_{new})$, i.e, $L_a(S) \subset L_a(S_{new})$.

Given any packet sequence $M \in L_a(S_{new})$.

- If M contains evasion packets corresponding to a vulnerable rule, $M \in L_s(S_{new})$ as the way S_{new} is constructed.

- If M does not contains evasion packets corresponding to any vulnerable rule, then $M \notin L_e(S)$. If $M \notin L_s(S_{new})$, then $M \notin L_s(S)$ because $L_s(S) \subset L_s(S_{new})$. So $M \notin L_a(S)$ because $M \notin L_e(S)$. Then $M \notin L_a(S_{new})$ because $L_a(S) \subset L_a(S_{new})$ which is a contradiction. Therefore $M \in L_s(S_{new})$.

Hence, there does not exists any packet sequence M so that $M \notin L_s(S_{new})$ and $M \in L_a(S_{new})$. So the new rule set is not vulnerable.

3.3.9 Implementation and Evaluation

We created a program called **SFET** (Snort *Flowbits* Evasion Tool) to parse a *flowbits* rule set, check if the rule set is vulnerable to the proposed evasion technique, generate the corresponding D_e (or evasion sequences) and patch the rule set accordingly depending on its size and the number of evasion sequences.

SFET can be run in 3 modes: specified mode, automatic mode and cautious mode. In the specified mode, **SFET** allows users to specify which rule is evadable and which rule is a target rule. In the automatic mode, **SFET** itself decides the possibility of a rule to be evadable based on the rule's matching options (like content options and traffic direction the rule matches) and chooses rules with no *flowbits:noalert* option as target rules. Lastly, in the cautious mode, **SFET** assumes all rules in a rule set are evadable and chooses only one rule from all possible target rules (no *flowbits:noalert* option) as the target rule. A rule set is considered vulnerable if there exists an evasion sequence for any chosen target rule.

We collected all possible rule sets publicly available from the internet (most of them from BleedingEdge [9] and SourceFire [42]). There are about 60% of the rules using *flowbits* that match traffic coming from the client's side (presumably from the attacker's side), hence these rules are considered evadable. All together (considering different rule options as well), there is about 68% out of the rules using *flowbits* determined by **SFET** evadable and highly evadable. In addition, there are about 6% and 4% out of 400 rule sets (using *flowbits*) detected vulnerable to the proposed attack when **SFET** was run in the cautious mode and the automatic mode respectively.

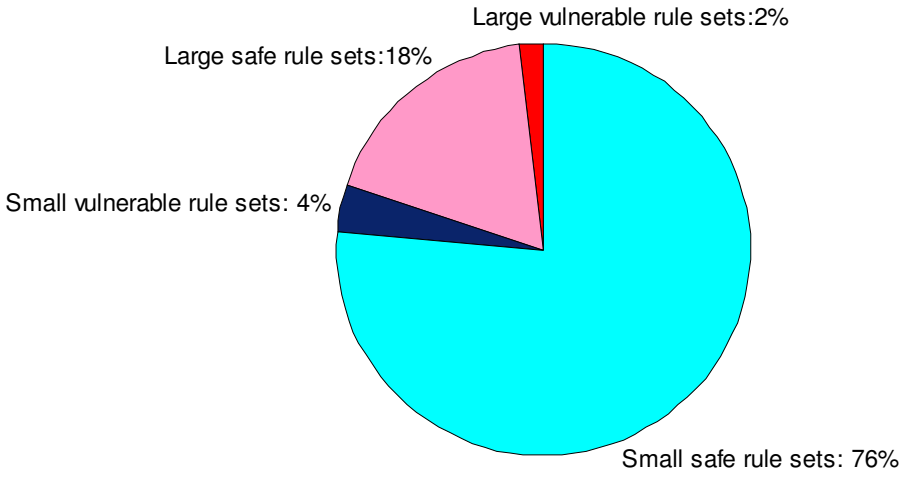


Figure 3.8: Vulnerable and safe rule sets percentage when SFET is run in the cautious mode

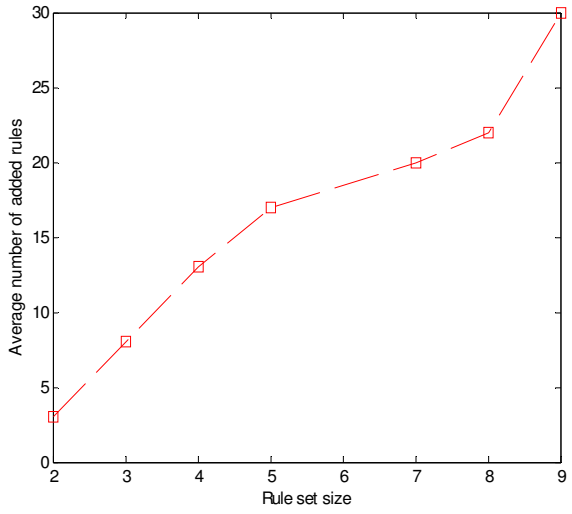


Figure 3.9: Overheads to patch small rule sets

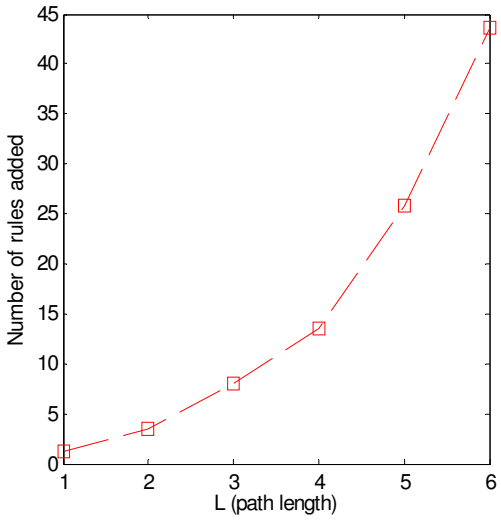


Figure 3.10: Overheads from false positives control patch

When running **SFET** in the specified mode with some chosen rule sets (we know exactly which rule is evadable), all evasion sequences generated by **SFET** can be converted to a real attack (this is not true for other modes).

Even though large rule sets (the number of rules is greater than 9) make up only 20% out of the considered rule sets, they are more susceptible to the attack than small rule sets. While 10% of large rule sets are vulnerable to the attack, only 5% of small rule sets are vulnerable. This is shown in Figure 3.8.

When applying the proposed solution to small vulnerable rule sets, the number of added rules to patch a rule set in average is triple the number of rules in the rule set (for both automatic and cautious modes). Figure 3.9 shows the average number of added rules for each rule set size (note: we do not find any vulnerable rule set of size 6).

For large vulnerable rule sets, the number of modified rules is the same as the number of vulnerable rules. Even though some large rule sets have many evadable rules, in average, only 10% of evadable rules are vulnerable. In addition, the number of added rules for each large rule set is as most the number of target rules in the rule set. The average number of added rules is only 3.5 for both automatic and cautious modes.

We chose some patched rule sets (including small and large rule sets) to apply the false positive control patch for different values of L. In average, the number of rules added to control false positives increases exponentially as L increases (as expected) and this is shown in Figure 3.10.

Chapter 4
Related work

In this chapter, we focus our study on the work that this thesis' contributions intersect with. Several research areas are closely related to what we have done in this thesis. Related to access control misconfigurations in general, Lujo Bauer et al. [5] applied association rule mining to the history of accesses to predict changes to access-control policies that are likely to be consistent with users' intentions, so that these changes can be instituted in advance of misconfigurations interfering with legitimate accesses. However, in this thesis, we only concentrate on NAC policies and we do not make use of the history of accesses to detect or predict misconfigurations. Instead, NAC policy misconfigurations are found from the policy itself. We provide tools to effectively support users in policy inspection (for firewall policies) and to automatically detect misconfigurations from a given policy (for Snort IDS policies).

Besides, the authors in [45] proposed an integrated, constraint-based approach for modeling and reasoning about firewall and NIDS policy configurations, however, they mainly deal with these components' misconfigurations when they are put together in a network and consider the dependencies among them to reason automatically about their combined behavior. Our work concentrates on each component and its policy separately.

Related to PolicyVis' design objectives, a significant amount of work has been reported in the area of network security visualization and firewall policy management.

There are many visualization tools introduced to enhance network security. PortVis [28] uses port-based detection of security activities and visualizes network traffic by choosing important features for axes and displaying network activities on the graph. SeeNet [6] supports three static network displays: two of these use geographical relationships, while the third is a matrix arrangement that gives equal emphasis to all network links. NVisionIP [23] uses a graphical representation of a class-B network to allow users to quickly visualize the current state of networks. Le Malécot et al. [27] introduced an original visualization design which combines 2-D and 3-D visualization for network traffic monitoring. However, these tools only focus on visualizing network traffic to assist users in understanding network events and taking according actions.

Moreover, previous work on firewall visualization only concentrates on visualizing how firewalls react to network traffic based on network events. Chris P. Lee et al. [25] proposed a tool visualizing firewall reactions to network traffic to aid users in configuration of firewalls. FireViz [38] visually

displays activities of a personal firewall in real time to possibly find any potential loop holes in the firewall's security policies. These tools can only detect a small subset of all firewall behaviors and can not determine all possible potential firewall patterns by looking at the policy directly like PolicyVis. Besides, Tufin SecureTrack [46] is a commercial firewall operations management solution; however, it provides change management and version control for firewall policy update. It basically visualizes firewall policy version changes, but not rule properties and relations and allows users to receive alerts if any change violates the organizational policy. Thus, Tufin SecureTrack cannot be used for rules analysis and anomaly discovery. Finally, Firmato [50], a firewall management toolkit, has a graphical firewall rule illustrator. This graphical component visualizes both the host-group's structure and the services (packets) that the firewall passes. However, it does not allow the user to visualize chosen firewall scopes and firewall anomalies.

In the field of firewall policy management, a filtering policy translation tool proposed in [14] describes, in a natural textual language, the meaning and the interactions of all filtering rules in the policy, revealing the complete semantics of the policy in a very concise fashion. However, this tool is not as efficient as PolicyVis in helping users capture the policy properties quickly in case of huge number of rules in the policy. In [15], the authors mentioned firewall policy anomalies and techniques to discover them, and suggested a tool called Firewall Policy Advisor which implements anomaly discovery algorithms. However, Firewall Policy Advisor is not capable of showing all potential behaviors of firewall policies and does not help users in telling if a policy does what he wants.

The authors in [16] [4] suggest methods for detecting and resolving conflicts among general packet filters. However, only correlation anomaly [15] is considered because it causes ambiguity in packet classifiers. In addition, the authors in [3] [49] [2] proposed firewall analysis tools that allow users to issue customized queries on a set of filtering rules and display corresponding outputs in the policy. However, the query reply could be overwhelming and still complex to understand. PolicyVis output is more comprehensible. Moreover, those tools require users to consider very specific issues to inspect the policy. PolicyVis, on the other hand, enables users to investigate the policy at once, which is more practical and efficient in large policies.

In this thesis, the work on Snort policy misconfigurations intersects with two research areas: NIDS policy management and NIDS attack and evasion techniques.

IDS Policy Manager [21] is a tool to manage Snort IDS sensors in a distributed environment. It helps deploy and update Snort policies to sensors in an easy manner; however, it is not capable of detecting any misconfiguration in Snort policies. In fact, there has not been much work done in the

area of NIDS policy misconfigurations. To the best of our knowledge, we are the first to classify policy misconfigurations in an NIDS.

In the field of IDS attacks and evasion techniques, even though there are attacks on host-based IDS like mimicry attacks introduced by Wagner and Soto [48], attacks and evasion techniques on NIDS seems to be a more interesting topic and have been very well studied in the literature. Ptacek and Newsham [44] were the first to bring up a way to evade a NIDS by using TCP Segmentation and IP Fragmentation, and FragRoute is the tool created to carry out these evasion techniques. A NIDS needs to carry out TCP segments and IP fragments reassembly to defend these evasion techniques. Besides, Handley and Paxson [20] [32] discussed evasion techniques based on inherent ambiguities of the TCP/IP protocol which leads to the difference between a NIDS and its protected system in performing TCP segments and IP fragments reassembly. Traffic normalization suggested by Handley et al. [20] tries to remove these ambiguities by patching the packet stream. Another solution to this was proposed by Shankar and Paxson [37], Active Mapping, which eliminates TCP/IP-based ambiguity in a NIDS' analysis with minimal runtime cost. Active Mapping efficiently builds profiles of the network topology and the TCP/IP policies of hosts on the network; a NIDS may then use the host profiles to disambiguate the interpretation of the network traffic on a per-host basis. This idea is implemented in the Stream5 [22] preprocessor of Snort, which makes Snort a "target-based" NIDS.

Snot [40], Stick [17], IDSWakeup [36] and Mucus [29] are over-stimulation tools that cause a DOS attack to Snort by trying to overload Snort with alerts from mutated packets constructed from Snort rules. Another DOS attack to a NIDS comes from the algorithmic complexity issue [12] [39] especially the authors in [39] presented a highly effective attack against Snort, and provided a practical algorithmic solution that successfully thwarts the attack.

Rubin et al. [34] observed that different attack instances can be derived from each other using simple transformations. TCP and application-level transformations are modeled as inference rules in a natural-deduction system. Starting from an exemplary attack instance, they used an inference engine to automatically generate all possible instances derived by a set of rules. They created AGENT, a tool capable of both generating attack instances for NIDS testing and determining whether a given sequence of packets is an attack. However, each attack generated by our evasion technique against the *flowbits* misconfiguration is neither a TCP nor an application-level transformation, hence, is not an instance generated by AGENT, assuming that the rule set represents the original exemplary attack instance. Although in this thesis, we only deal with Snort rules, which are mainly manually written by

users, automatic generated semantic-aware signatures [51] or session signatures [35] are still possibly vulnerable to our proposed evasion technique. In order to avoid false positives, these generated signatures must consider "innocent" paths (or sequences) which are not attack instances. Our evasion technique exploits these "innocent" paths and tries to convince Snort that the session is following one of the "innocent" paths.

Chapter 5
Conclusion and Future Work

5.1 Conclusion

NAC systems provide proper security services if they are correctly configured and efficiently managed. NAC policies used in enterprise networks are getting more complex as the number of rules and devices becomes larger. This significantly increases the possibility of policy misconfigurations and network vulnerabilities. As a result, it is essential that we need to classify and detect policy misconfigurations of any NAC system. In addition, it will be very useful if there is an effective policy management tool which significantly helps users in discovering NAC policy misconfigurations. In this thesis, we chose to study and analyze the NAC policy configuration of two significant network security devices, namely, firewall and IDS/IPS.

In the first part of the thesis, we presented PolicyVis, a tool that visualizes firewall rules to efficiently enhance the understanding and inspection of firewall configuration. PolicyVis provides visual views on firewall policies and rules, which gives users a powerful means for inspecting firewall policies. In this thesis, we described design features of PolicyVis and illustrated PolicyVis with multiple examples showing the effectiveness and usefulness of PolicyVis in determining the policy behavior in various case studies. We presented concepts and techniques to find rule anomalies in PolicyVis. Besides, we also showed how PolicyVis visualizes distributed firewalls to achieve same benefits as visualizing single firewalls. Finally, we presented the implementation and evaluation of PolicyVis.

Using PolicyVis was shown very effective for firewalls in real-life networks. In regards to usability, unskilled people with short time of learning how to use PolicyVis can quickly understand and start using all features of PolicyVis. Moreover, by evaluation, PolicyVis effectively helped users including network security juniors and seniors to figure out firewall policy behavior easily by reviewing the visualizing of primitive firewall rules. In addition, PolicyVis was shown a very good tool in finding rule anomalies or conflicts easily and quickly. The number of dimensions users need to consider during firewall inspection varies according to situations; however, considering all possible rule fields always gives users a better analysis of the policy.

In the second part of the thesis, we classified and gave examples of various policy misconfigurations of Snort. Then we particularly concentrated on a class of *flowbits* misconfiguration that makes Snort susceptible to false negatives. We formalized the *flowbits* misconfiguration problem

and suggested practical solutions to patch small and large vulnerable rule sets. Moreover, we formally proved that the solutions are complete and sound. In addition to these solutions, we proposed a false positives control method which allows Snort users to reduce potential false positives caused by the patches.

We implemented a tool called **SFET,** which can automatically calculate the possibility that a rule is evadable based on the rule's content and generate all possible evasion sequences to a given *flowbits* rule set. Besides, **SFET** is also able to augment the rule set with additional rules that thwart the proposed attack without changing the functionality of the original rule set.

The evaluation showed that a large number of available *flowbits* rule sets are vulnerable, which seriously affects the security of Snort users' systems. However, as shown, the practical nature of our solutions generates little overhead for both small and large vulnerable rule sets.

5.2 Future work

This thesis is limited to the study and analysis of firewall and IDS/IPS policy misconfigurations, therefore we plan to work on policy misconfigurations of other NAC security systems in the near future.

Related to PolicyVis, even though the tool was shown a very effective tool, we still want to perform more evaluation on it and collect more users' ideas to make PolicyVis a better tool for the inspection of firewall policy misconfigurations. There are still many possible features that we want to implement in PolicyVis to maximize its usability as well as efficiency. We want PolicyVis to support more viewing levels of firewall policies and automatically show users possible strange behaviors (possibly misconfigurations) and true rule anomalies of firewall policies on the graph. In addition, PolicyVis currently shows how to visualize stateless firewalls but we can easily envision extending this to visualize stateful firewalls too by preprocessing the policy to create the implicit rules in stateful firewalls. We consider supporting stateful firewalls in PolicyVis in our future work.

Related to the work on Snort rule misconfigurations, the first thing we want to accomplish in the future is to provide detection methods and solutions for all of the classified Snort rule misconfigurations. Moreover, we plan to extend the concept of our defined Snort rule misconfigurations (including the *flowbits* misconfiguration and the evasion technique) to different NIDSs as well. Even though we have proposed a method to control false positives in addition to the solutions for vulnerable *flowbits* rule sets, accurately measuring how much false positives caused by

the patched rule sets is still nesseccary. However, we leave this task for future work because it requires Snort to run for a long period of time in order to have a correct answer.

Appendix A

The new rule set is complete (solution for small rule sets)

Lemma 1: If $S = S_1 \cup S_2$, $L_s(S) = L_s(S_1) \cup L_s(S_2)$.

Proof:

Because $S = S_1 \cup S_2$, so target rules of S = target rules of S_1 $\cup$ target rules of S1. Because S_1 and S_2 are independent, then if a packet sequence puts S_1 in a target state, it still puts S_1 in a target state when S_2 is added to S_1. The same argument applies for S_2. A target state of S is a state where a target rule of S can be triggered, which is either S_1's target rule or S_2's target rule. It means that a target state of S is either S_1's target state or S_2's target state. Therefore, a packet sequence that puts S in a target state if and only if it puts S_1 or S_2 in a target state. So we have $L_s(S) = L_s(S_1) \cup L_s(S_2)$.

Lemma 2: If $S = S_1 \cup S_2$, $L_a(S) = L_a(S_1) \cup L_a(S_2)$.

Proof:

We'll prove that $D_a(S) = D_a(S_1) \cup D_a(S_2)$

- By Lemma 1, we have $L_s(S) = L_s(S_1) \cup L_s(S_2)$. So $D_s(S) = D_s(S_1) \cup D_s(S_2)$.
- Let $D^* = D_a(S_1) \cup D_a(S_2)$.
- Let $F_{s\text{-}S}$, $F_{s\text{-}S1}$, $F_{s\text{-}S2}$, $F_{a\text{-}S}$, $F_{a\text{-}S1}$, $F_{a\text{-}S2}$, and F^* be transition functions for $D_s(S)$, $D_s(S_1)$, $D_s(S_2)$, $D_a(S)$, $D_a(S_1)$, $D_a(S_2)$, and D^* respectively.
- Both D* and $D_a(S)$ have the same alphabet which is $\cup\{P_i, P_i^*$(if R_i is evadable) : R_i in S and R_i is not a target rule$\}$
- Given any state A in $D_s(S_1)$, any state B in $D_s(S_2)$, any R_i in S:

+ By the way union of two Ds is constructed, the transition function of $D_s(S)$ is as follow:

$F_{s\text{-}S}([A,B],x) = [F_{s\text{-}S1}(A,x) , F_{s\text{-}S2}(B,x)]$

+ By the way union of two Ds is constructed, the transition function of D* is as follow:

$F^*([A,B],x) = [F_{a\text{-}S1}(A,x) , F_{a\text{-}S2}(B,x)]$

+ Based on the construction of D_s and D_a of a rule set, we have:

$F_{a\text{-}S}([A,B],x) = F_{s\text{-}S}([A,B],x)$ if x is P_i

$F_{a\text{-}S}([A,B],x) = [A,B]$ if x is P_i^* and R_i is evadable

$F_{a\text{-}S1}(A,x) = F_{s\text{-}S1}(A,x)$ if x is P_i

$F_{a\text{-}S1}(A,x) = A$ if x is P_i^* and R_i is evadable

$F_{a\text{-}S2}(B,x) = F_{s\text{-}S2}(B,x)$ if x is P_i

$F_{a\text{-}S2}(B,x) = B$ if x is P_i^* and R_i is evadable

- We have:

+ if x is P_i:

$F_{a\text{-}S}([A,B],x) = F_{s\text{-}S}([A,B],x) = [F_{s\text{-}S1}(A,x) , F_{s\text{-}S2}(B,x)] = [F_{a\text{-}S1}(A,x) , F_{a\text{-}S2}(B,x)] = F^*([A,B],x)$

+ if x is P_i^* and R_i is evadable:

$F_{a\text{-}S}([A,B],x) = [A,B] = [F_{a\text{-}S1}(A,x) , F_{a\text{-}S2}(B,x)] = F^*([A,B],x)$

→ $F^* = F_{a\text{-}S}$

- Let $A_{s\text{-}S}$, $A_{s\text{-}S1}$, $A_{s\text{-}S2}$, $A_{a\text{-}S}$, $A_{a\text{-}S1}$, $A_{a\text{-}S2}$, and A^* be sets of accept states of $D_s(S)$, $D_s(S_1)$, $D_s(S_2)$, $D_a(S)$, $D_a(S_1)$, $D_a(S_2)$, and D^* respectively. We have:

$A_{a\text{-}S} = A_{s\text{-}S}$ (the construction of D_s and D_a) and $A_{s\text{-}S} = \{[e_1, e_2]$ in which either e_1 is in $A_{s\text{-}S1}$ or e_2 is in $A_{s\text{-}S2}\}$ (the construction of union of two Ds)

$A^* = \{[e_1, e_2]$ in which either e_1 is in $A_{a\text{-}S1}$ or e_2 is in $A_{a\text{-}S2}\} = \{[e_1, e_2]$ in which either e_1 is in $A_{s\text{-}S1}$ or e_2 is in $A_{s\text{-}S2}\}$

→ $A_{a\text{-}S} = A^*$

- With the same argument, we also have that $D_a(S)$ and D* have the same start state.

=> We have shown that $D_a(S)$ and D* have the same alphabet, state transition functions, start state and accept states. So $D_a(S) = D^* = D_a(S_1) \cup D_a(S_2)$, in other words, $L_a(S) = L_a(S_1) \cup L_a(S_2)$.

A.1 The new rule set can detect all packet sequences detected by the old rule set

Proof:

We have $S_{new} = S \cup S_1 \cup S_2 \ldots \cup S_n$ where S, S_1, $S_2 \ldots$,S_n use non-overlapping label sets. So by Lemma 1,

$L_s(S_{new}) = L_s(S \cup S_1 \cup S_2 \ldots \cup S_n) = L_s(S) \cup L_s(S_1 \cup S_2 \ldots \cup S_n)$ (because S and $S_1 \cup S_2 \ldots \cup S_n$ use non-overlapping label sets and are independent)

→ All packet sequences detected by S are detected by S_{new}.

A.2 The new rule set can detect all possible evasion sequences

Proof:

- We need to prove that, given any path X from the start state to an accept state in the $D_e(S)$, the new rule set can detect the packet sequence corresponding to X, say X_{es}.

- Let Z be the simple path after all circles are removed from X. Let Y be the simple path in ***SP*** such that Y's corresponding packet sequence is a prefix of Z's corresponding packet sequence. Let S_Y be

the rule set added for Y and Y_{es} be Y's corresponding packet sequence. We will prove that X_{es} is detected by S_Y or accepted by $D_S(S_Y)$. Assume S_Y uses label set $\{A_1, A_2,\ldots,A_n\}$ and $Y_{es} = Y_1\ Y_2\ \ldots.\ Y_{n-1}$. So $X_{es} = Y_1\ X_1\ Y_2\ X_2\ \ldots\ Y_{n-2}\ X_{n-2}\ Y_{n-1}X_{n-1}$ where X_i is any packet sequence.

- Let T_i be the state where $A_1,\ldots,A_i$ are set and $A_{i+1},\ldots,A_n$ are not set. Let F_s be the transition function of $D_s(S_Y)$.

- By the construction of S_Y, we have

$F_s(T_i, Y_J) = T_{i+1}$ if i=j and T_i if i!=j (i<n)

$F_s(T_i, x) = T_i$ if x != Y_J for some j or i=n

→ $F_s(T_i, X_J) = T_k$ where k>=i

- Now let's consider how X_{es} is processed by $D_s(S_Y)$:

We'll prove that the state right before $D_s(S_Y)$ sees Y_i is T_c where c>=i. This can be proven by using induction.

It is true for i=1. Assume it is true for 1<=i<=k.

Let T_e and T_f be the states right before and after $D_s(S_Y)$ sees Y_k repsectively. We know e>=k because of the inductive hypothesis.

If e=k, $T_f = F_s(T_k, Y_k) = T_{k+1}$ → f=k+1>k.

If e>k, $T_f = F_s(T_e, Y_k) = T_e$ → f=e >k.

So f>=k+1

We have $T_w = F_s(T_f, X_k)$ (w>=f) is the state right before $D_s(S_Y)$ sees Y_{k+1}. We have w>=f>=k+1.

So we have proven that the state right before $D_s(S_Y)$ sees Y_i is T_c where c>=i. It means that the state right before $D_s(S_Y)$ sees Y_{n-1} is T_{n-1} or T_n. So $F_s(T_1, X_{es}) = F_s(T_{n-1}, Y_{n-1}X_{n-1})=T_n$ or $F_s(T_n, Y_{n-1}X_{n-1})= T_n$.

→ $F_s(T_1, X_{es}) = T_n$.

- So X_{es} is accepted by $D_s(S_Y)$ and hence detected by S_Y.

- Since S_Y is independent with the rest in the new rule set (because S_Y uses a different label set), a packet sequence detected by S_Y is detected by the new rule set (by Lemma 1)

Appendix B

The new rule set is sound (solution for small rule sets)

Lemma 3: Each added rule set corresponding to a simple path is not vulnerable to the proposed attack.

Proof:

Assume the rule set S is added for the simple path K. We need to show that $L_e(S) = \emptyset$.

Assume $S=\{R_1,R_2,\ldots,R_n\}$ and uses label set $\{A_1, A_2,\ldots,A_n\}$.Let T_i be the state where $A_1,\ldots,A_i$ are set and $A_{i+1},\ldots,A_n$ are not set.

Let F_s , F_a, and F_e be transition functions for $D_s(S)$, $D_a(S)$, and $D_e(S)$ respectively.

- Based on the construction of S from K, we have

$F_s(T_i, P_J) = T_{i+1}$ if i=j and T_i if i!=j (i<n)

$F_s(T_i, P_J^*) = T_{i+1}$ if i=j and T_i if i!=j (If R_J is evadable) (i<n)

$F_s(T_n, x) = T_n$

$F_a(T_i, P_J) = T_{i+1}$ if i=j and T_i if i!=j (i<n)

$F_a(T_i, P_J^*) = T_i$ (If R_J is evadable) (i<n)

$F_a(T_n, x) = T_n$

- Based on the construction of $D_e(S)$, we have:

+ If i=j:

$F_e([T_i,T_J], x) = F_e([T_i,T_i], x) = [F_s(T_i, x), F_a(T_i, x)] = [T_i, T_i]$ or $[T_{i+1}, T_{i+1}]$ or $[T_{i+1},T_i]$

+ If j<i:

$F_e([T_i,T_J], x) = [F_s(T_i, x), F_a(T_J, x)] = [T_i, T_J]$ or $[T_i, T_{J+1}]$ or $[T_{i+1},T_J]$

Initially, the start state is $[T_1, T_1]$, so based on the transition functions above, it is clear that $[T_i , T_J]$ (j>i) is not reachable in the $D_e(S)$

However, accept states of $D_e(S)$ are $[T_i,T_n]$ where 1<= i < n , so $L_e(S) = \emptyset$.

Therefore, each added rule set corresponding to a simple path is not vulnerable to the proposed attack.

The new rule set itself is not vulnerable to the proposed attack

Proof :

Assume $S_1, S_2 \ldots, S_n$ are added rule sets for n simple paths in ***SP***

We have $S_{new} = S \cup S_1 \cup S_{2\ldots} \cup S_n$ where $S, S_1, S_{2\ldots}, S_n$ use non-overlapping label sets.

We need to prove that the new rule set S_{new} is safe against the proposed attack. In other words, we need to prove that $L_e(S_{new}) = \emptyset$.

From what we have proved in Appendix A.2, we have: $L_e(S) = L_s(S_1) \cup L_s(S_2) \cup \ldots \cup L_s(S_n)$ (*)

Now, we have $L_e(S_{new}) = L_e(S \cup S_1 \cup S_{2\ldots} \cup S_n) = \neg L_s(S \cup S_1 \cup S_{2\ldots} \cup S_n) \cap L_a(S \cup S_1 \cup S_{2\ldots} \cup S_n)$

$= (\neg L_s(S) \cap \neg L_s(S_1) \cap \ldots \cap \neg L_s(S_n)) \cap (L_a(S) \cup L_a(S_1) \cup L_a(S_{2\ldots}) \cup L_a(S_n))$ (by Lemma 1 and Lemma 2)

$= (\neg L_s(S) \cap \neg L_s(S_1) \cap \ldots \cap \neg L_s(S_n) \cap L_a(S)) \cup (\neg L_s(S) \cap \neg L_s(S_1) \cap \ldots \cap \neg L_s(S_n) \cap L_a(S_1)) \cup \ldots.$

$\cup (\neg L_s(S) \cap \neg L_s(S_1) \cap \ldots \cap \neg L_s(S_n) \cap L_a(S_n))$

$= (\neg L_s(S) \cap \neg L_s(S_1) \cap \ldots \cap \neg L_s(S_n) \cap L_a(S)) \cup (\neg L_s(S) \cap \ldots \cap \neg L_s(S_n) \cap L_e(S_1)) \cup \ldots. \cup$

$(\neg L_s(S) \cap \ldots \cap \neg L_s(S_{n-1}) \cap L_e(S_n))$

$= (\neg L_s(S) \cap \neg L_s(S_1) \cap \ldots \cap \neg L_s(S_n) \cap L_a(S)) \cup \emptyset \ldots \cup \emptyset$ (by lemma 3)

$= \neg L_s(S) \cap \neg L_s(S_1) \cap \ldots \cap \neg L_s(S_n) \cap L_a(S)$

$= (\neg L_s(S) \cap L_a(S)) \cap \neg L_s(S_1) \cap \ldots \cap \neg L_s(S_n)$

$= L_e(S) \cap \neg L_s(S_1) \cap \ldots \cap \neg L_s(S_n)$

$= (L_s(S_1) \cup L_s(S_2) \cup \ldots \cup L_s(S_n)) \cap \neg L_s(S_1) \cap \ldots \cap \neg L_s(S_n)$ (by (*))

$= \emptyset$ (use induction on *n* and use $L_s(S_i) \cap \neg L_s(S_i) = \emptyset$)

So we have $L_e(S_{new}) = \emptyset$, this means S_{new} is not vulnerable to the proposed attack.

Appendix C

C.1 Theorem: If a set Q of evadable rules makes the rule set vulnerable, then at least one of these evadable rules is vulnerable.

Proof:

We'll prove this theorem by contradiction.

Assume there is no rule in the set is vulnerable. Let K be an evasion sequence that exploits these evadable rules in Q. K contains evasion packets correponding to evadable rules in Q and normal packets.

So $K \notin L_s(S)$ and $K \in L_a(S)$.

Let K^a be the packet sequence from K without evasion packets (removing evasion packets from K), so $K^a \in L_a(S)$.

Let K^f be the packet sequence from K with P_i*'s replaced by P_i's for all $R_i \in Q$. We have $K^f \notin L_s(S)$ because from Snort's perspective, K and K^f are the same, and we also have $K \notin L_s(S)$

Now we consider this procedure:

Let $K^s = K^a$

For each $R_i \in Q$:

1- Set $K^a = K^s$

2- Add P_i*'s to K^a at the positions where they are removed from K

3- Set K^s = packet sequence from K^a with P_i*'s replaced by P_i's.

Before the first iteration: $K^a \in L_a(S)$

After the first iteration: We have $K^a \in L_a(S)$ because we just add evasion packets to K^a. We also have K^a contains evasion packets correnspounding to only R_i for some i. So K^a can not be an evasion sequence, otherwise R_i is vulnerable. Together we must have $K^a \in L_s(S)$. But from Snort's perspective, K^a and K^s are the same, so $K^s \in L_s(S)$. Since K^s does not contain any evasion packet, then if $K^s \in L_s(S) \rightarrow K^s \in L_a(S)$.

With the same arguments, we always have $K^s \in L_s(S)$ after each iteration. But when the loop is done, we have $K^s = K^f$. This means $K^f \in L_s(S)$: a contradiction.

So the assumption is wrong. Therefore, there is at least one rule in the set is vulnerable.

C.2 Small rule set solution complexity proof

In D_e , a simple path from the start node to a target node can be of length M^2. There is one choice for the start node in the path, |Σ| choices for the second node, |Σ|^2 for the third, … and so on until node M^2-|T|, entailing an upper bound complexity of ~ |Σ|^(M^2-|T|-1) for each target state in |T|.

For each simple path in *SP* of size |p|, a set of |p|-1 flowbits and rules is created. |p| has an upper bound length of M^2-|T|.

In summary, an upper bound on the complexity of the procedure is |T| x (M^2-|T|-1) x |Σ|^(M^2-|T|-1) new rules!

C.3 Large rule set solution complexity proof

We always have |Σ| <= |R|.

Number of edges in D_s= M x |Σ|.

The evasion DFAconstruction has a worst case cost of M^2 x |Σ|.

For each constructed D_e, there is a need to run an algorithm to find if a final state is reachable from the start state. This can be avoided by doing this test while constructing the D_e, so the this operation can be assumed to have no cost.

The step of adding and changing rules to the rule set has an upper bound of |R| operations.

In total, the worst case complexity is |R| x (M^2 x |Σ| + 1), which is polynomial.

Bibliography

[1] A. El-Atawy, K. Ibrahim, H. Hamed, and E. Al-Shaer, "Policy segmentation for intelligent firewall testing," in *1st Workshop on Secure Network Protocols*, November 2005.

[2] A. Mayer, A. Wool, and E. Ziskind, "Fang: A firewall analysis engine," in *Proc. IEEE Symp. on Security and Privacy*, Oakland, CA, May 2000, pp. 177-187.

[3] A. X. Liu, M. G. Gouda, H. H. Ma, and A. H. Ngu, "Firewall queries," in *Proceedings of the 8th International Conference on Principles of Distributed Systems*, December 2004.

[4] B. Hari, S. Suri and G. Parulkar, "Detecting and Resolving Packet Filter Conflicts," in *Proceedings of IEEE INFOCOM '00*, March 2000.

[5] L. Bauer, S. Garriss, and M. K. Reiter, "Detecting and resolving policy misconfigurations in access-control systems," in *SACMAT '08: Proceedings of the 13th ACM Symposium on Access Control Models and Technologies*, New York, NY, USA, 2008, pp. 185-194.

[6] R. A. Becker, S. G. Eick, and A. R. Wilks, "Visualizing Network Data," *IEEE Transactions on Visualization and Computer Graphics*, vol. 1, no. 1, pp. 16-28, 1995.

[7] M. Benantar, *Access Control Systems: Security, Identity Management and Trust Models*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2005.

[8] Bethel, E. W., S. Campbell, E. Dart, K. Stockinger, and K. Wu, "Accelerating Network Traffic Analysis Using Query-Driven Visualization," in *IEEE Symposium on Visual Analytics Science and Technology , IEEE Computer Society Press*, 2006.

[9] BleedingEdge Inc. [Online]. http://www.bleedingthreats.net

[10] B. Caswell, J. Beale, and A. R. Baker, *Snort Intrusion Detection and Prevention Toolkit*.: Syngress Publishing, 2007.

[11] Cisco Systems, Inc. (2006-2008) Installing and Using Cisco Intrusion Prevention System Device Manager 6.0. OL-8824-01.

[12] S. A. Crosby and D. S. Wallach, "Denial of service via algorithmic complexity attacks," in *SSYM '03: Proceedings of the 12th Conference on USENIX Security Symposium*, Berkeley, CA, USA, 2003, pp. 3-3.

[13] David Gilbert and Thomas Morgner. JFreeChart, Java chart library. [Online]. http://www.jfree.org/jfreechart/

[14] E. Al-Shaer and Hazem Hamed, "Discovery of Policy Anomalies in Distributed Firewalls," in *Proceedings of IEEE INFOCOM '04*, March 2004.

[15] Ehab Al-Shaer, Hazem Hamed, Raouf Boutaba, and Masum Hasan, "Conflict classification and analysis of distributed firewall policies," *IEEE Journal on Selected Areas in Communications*, pp. 23(10):2069--2084, October 2005.

[16] D. Eppstein and S. Muthukrishnan, "Internet packet filter management and rectangle geometry," in *SODA '01: Proceedings of The Twelfth Annual ACM-SIAM Symposium on Discrete Algorithms*, Philadelphia, PA, USA, 2001, pp. 827-835.

[17] C. Giovanni. (2001, March) Fun with Packets: Designing a Stick, Draft White Paper on Stick, Tech. Rep. [Online]. http://www.eurocompton.net/stick

[18] J. R. Goodall, W. G. Lutters, P. Rheingans, and A. Komlodi, "Preserving the Big Picture: Visual Network Traffic Analysis with TN," in *VIZSEC '05: Proceedings of the IEEE Workshops on Visualization for Computer Security*, Washington, DC, USA, 2005, p. 6.

[19] H. Hamed, E. Al-Shaer, and W. Marrero, "Modeling and Verification of IPSec and VPN Security Policies," in *ICNP '05: Proceedings of the 13TH IEEE International Conference on Network Protocols*, Washington, DC, USA, 2005, pp. 259-278.

[20] M. Handley, V. Paxson, and C. Kreibich, "Network intrusion detection: evasion, traffic normalization, and end-to-end protocol semantics," in *SSYM '01: Proceedings of The 10th Conference on USENIX Security Symposium*, Berkeley, CA, USA, 2001, pp. 9-9.

[21] IDS Policy Manager. [Online]. http://www.activeworx.org/Default.aspx?tabid=55

[22] J. Novak and S. Sturges, "Target-Based TCP Stream Reassembly," Aug 2007.

[23] K. Lakkaraju, R. Bearavolu, and W. Yurcik, "NVisionIP - a traffic visualization tool for large and complex network systems," in *International Multiconference on Measurement, Modelling, and Evaluation of Computer-Communication Systems (Performance TOOLS)*, 2003.

[24] B. W. Lampson, "Protection," in *Proceedings of the 5th Princeton Symposium on Information Science and Systems*, March 1971, pp. 437--443.

[25] C. P. Lee, J. Trost, N. Gibbs, R. Beyah, and J. A. Copeland, "Visual Firewall: Real-time Network Security Monito," in *VIZSEC '05: Proceedings of the IEEE Workshops on Visualization for Computer Security*, Washington, DC, USA, 2005, p. 16.

[26] J. Lind-Nielsen. The buddy obdd package. [Online]. http://www.bddportal.org/buddy.html

[27] E. L. Malécot, M. Kohara, Y. Hori, and K. Sakurai, "Interactively combining 2D and 3D visualization for network traffic monitoring," in *VizSEC '06: Proceedings ofTthe 3rd International Workshop on Visualization for Computer Security*, New York, NY, USA, 2006, pp. 123-127.

[28] J. McPherson, K.-L. Ma, P. Krystosk, T. Bartoletti, and M. Christensen, "PortVis: a tool for port-based detection of security events," in *VizSEC/DMSEC '04: Proceedings of the 2004 ACM Workshop on Visualization and Data Mining for Computer Security*, New York, NY, USA, 2004, pp. 73-81.

[29] D. Mutz, G. Vigna, and R. Kemmerer, "An Experience Developing an IDS Stimulator for the Black-Box Testing of Network Intrusion Detection Systems," in *ACSAC '03: Proceedings of the 19th Annual Computer Security Applications Conference*, Washington, DC, USA, 2003, p. 374.

[30] Network Access Control. [Online]. http://en.wikipedia.org/wiki/Network_Access_Control

[31] Nmap. [Online]. http://nmap.org

[32] V. Paxson, "Bro: a system for detecting network intruders in real-time," in *SSYM '98: Proceedings of The 7th Conference on USENIX Security Symposium*, Berkeley, CA, USA, 1998, pp. 3-3.

[33] M. Roesch, "Snort - Lightweight Intrusion Detection for Networks," in *LISA '99: Proceedings of The 13th USENIX Conference on System Administration*, Berkeley, CA, USA, 1999, pp. 229-238.

[34] S. Rubin, S. Jha, and B. P. Miller, "Automatic Generation and Analysis of NIDS Attacks," in *ACSAC '04: Proceedings of the 20th Annual Computer Security Applications Conference*, Washington, DC, USA, 2004, pp. 28-38.

[35] S. Rubin, S. Jha, and B. P. Miller, "Language-Based Generation and Evaluation of NIDS Signatures," in *SP '05: Proceedings of the 2005 IEEE Symposium on Security and Privacy*, Washington, DC, USA, 2005, pp. 3-17.

[36] S.Aubert. IDSWakeup. [Online]. http://www.hsc.fr/ressources/outils/ids wakeup/, 2000

[37] U. Shankar and V. Paxson, "Active Mapping: Resisting NIDS Evasion without Altering Traffic," in *SP '03: Proceedings of the 2003 IEEE Symposium on Security and Privacy*, Washington, DC, USA, 2003, p. 44.

[38] Sharma Nidhi, "FireViz : a personal firewall visualizing tool," Massachusetts Institute of

Technology, Dept. of Electrical Engineering and Computer Science, Thesis (M. Eng.) 2005.

[39] R. Smith, C. Estan, and S. Jha, "Backtracking Algorithmic Complexity Attacks against a NIDS," in *ACSAC '06: Proceedings of the 22nd Annual Computer Security Applications Conference on Annual Computer Security Applications Conference*, Washington, DC, USA, 2006, pp. 89-98.

[40] Sniphs. (2003, January) Snot. [Online]. http://www.l0t3k.org/tools/IDS/snot-0.92a.tar.gz

[41] Snort 2.8.1. [Online]. http://www.snort.org/dl/current/snort-2.8.1.tar.gz

[42] SourceFire, Inc. [Online]. http://www.sourcefire.com

[43] Sourcefire, Inc. (2003-2008) The Snort Project.

[44] T. Ptacek and T. Newsham, "Insertion, Evasion, and Denial of Service: Eluding Network Intrusion Detection," , 1998.

[45] Tom'as E. Uribe and Cheung Steven, "Automatic analysis of firewall and network intrusion detection system configurations," in *FMSE '04: Proceedings of the 2004 ACM Workshop on Formal Methods in Security Engineering*, pp. 66--74.

[46] Tufin SecureTrack: Firewall Operations Management Solution. [Online]. http://www.tufin.com

[47] G. Vigna, W. Robertson, V. Kher, and R. A. Kemmerer, "A Stateful Intrusion Detection System for World-Wide Web Servers," in *ACSAC '03: Proceedings of the 19th Annual Computer Security Applications Conference*, Washington, DC, USA, 2003, p. 34.

[48] D. Wagner and P. Soto, "Mimicry attacks on host-based intrusion detection systems," in *CCS '02: Proceedings of the 9th ACM conference on Computer and Communications Security*, New York, NY, USA, 2002, pp. 255-264.

[49] A. Wool, "Architecting the Lumeta firewall analyzer," in *SSYM '01: Proceedings of the 10th Conference on USENIX Security Symposium*, Berkeley, CA, USA, 2001, pp. 7-7.

[50] Y. Bartal, A. Mayer, K. Nissim, and A. Wool, "Firmato: A novel firewall management toolkit," in *ACM Transactions on Computer Systems*, November 2004, pp. 22(4):381-420.

[51] V. Yegneswaran, J. T. Giffin, P. Barford, and S. Jha, "An architecture for generating semantics-aware signatures," in *SSYM '05: Proceedings of the 14th Conference on USENIX Security Symposium*, Berkeley, CA, USA, 2005, pp. 7-7.