A Simple Firewall Design

Advanced Algorithm Course
Stevens Institute of Technology, Hoboken NJ
Saeid Hosseinipoor, Yuandong Cyrus Liu, Dave Xian

1 Introduction

1.1 Problem Statement

The objective of this projects is design and implementation of a packet controller within a network traffic according to a set on given rules. The rules are provided in a rule file and specify whether a node and/or a sub network could pass a TCP/UDP request to the other node or subnetwork. The rule format is:

srcIP(s), destIP(s), Act

Where srcIP(s) is a single IP or a range of IPs which is sent the request, destIP(s) is a single IP or a range of IPs which accepts the request, and Act is the action should be applied which is Allow or Block.

We need to design the algorithm such that be able to check the redundant rules, and for the coming package, our algorithm should check the IP and return the action for the IP pairs according our rules file. Some rules may apply to a pair of source and destination IPs, if they imply the same action, it is safe to pick one of them. The problem rises when we have different action for a single pair of IPs. For this situation, we look at the assigned priority to each rule and pick the most important rule.

1.2 Assumptions

IPs are defined in standard formats of a.b.c.d which a-d are integer numbers between 0 and 255. Range of IPs described in standard form of a.b.c.* or a.b.c.d/e or similar standard forms. There is no guaranty of consistency in rules or avoiding redundancies. Since, we assume rules are added to the list with no consideration, we make another assumption that most recent rules have higher priority to apply. For sake of clarity, we assume that new rules added to the end of the available rule file. The rules on the top of the file, are older than rules in the bottom of the file.

A private repository on github was created for group members to ease up working on the project simultaneously. Python 3 was adopted as an implantation language.

2 Design

First of all, we need a data structure that is capable to fast store and access of IP addresses and range of the IP addresses. In order to define such data structure, we need to define an unique and useful form of IP addresses and ranges that could be easier to store or recover.

2.1 IP Translation

We know that an IP address is in form of d.d.d.d which d is an integer number between 0 and 255. Therefore we can translate each integer number into a 8-bit binary number. According to this mapping function we have a string of 32 bit of 0s and 1s.

$$IP \rightarrow \text{“}d_1 \dots d_{32}\text{“} \quad d_i \in \{0,1\}$$

For the range of IPs, we can use * as a wild char define whatever after fixed bits:

$$IP\text{-}Range \rightarrow \text{“}d_1 \dots d_k{*}\text{”} \quad d_i \in \{0,1\}$$

2.2 Data Structure

Inspiring by the hierarchy concept of IP addresses we define a binary search tree to store the IP address and ranges. The translation of the IP address and range into an IPString, helps us to define this binary tree, such that the left child of a node is the next 0 after current node, and the right child is the next 1 after current node.

Let's consider the root of the tree be "*" which is range of all IPs (*.*.*.*), the left child of root represent "0*" and the right child is "1*" and so on. In other words if we have a node of "$d_1 \dots d_k$*", the left child of this node is "$d_1 \dots d_k$0*", and the right child is "1$d_1 \dots d_k$1*". It is obvious that the parent of this node would be "$d_1 \dots d_{k-1}$*". As a result of this definition, leaves are single IP address and intermediates nodes are IP ranges.

The maximum height of this tree is 32 which is the maximum length of the IPString. Access to the nodes in the worst case is 32 operation which is O(1).

3 Implementation

We need to implement different operations like insertion, search, match, and update in this project. More operations might be added if they are required during the development phase.

3.1 Insertion

After IP translation we start from root and traverse the tree to find the source IP(s)'s position in the tree. If the children are available we go through the nodes until get the target node, then insert the rule into a table or link list of the rules. In such case that children are not available we make new child in proper position and open a new list or table of rules in the final node.

We investigated to consider two tree for source and destination IPs, but analysis shows that it doesn't improve the performance.

3.2 Search

IPString, translated IP address or range, is a unique representation of various form of IP address or ranges. We find the source IP as described in the insertion operation, then retrieve the rule table. We pick all the rules from the nodes in the path from root to source IP node which matches the destination IP because all the parent rules should be applied to the traffic packet. The most important rule, based on priority, will be applied to the traffic packet.

3.3 Update

Similar to insertion operation, we will find the position of the rule. The difference is that we add the rule to the node if its priority is higher than the redundant rule or there is no redundant rule in the rule list ignore it. This strategy might be updated and improved in future.

3.4 Match

This is boolean function that check whether an IP address (e.g. destination address) in equal to the given IP address or is in the range of a given IP range. In other words, it checks if a given IPString covers or equals to other IPString.

4 Analysis

4.1 Time Complexity

The time complexity of creation of the data list is O(n), because we need to add, insert each n rules in the data structure in constant time. However the complexity for search and reaching the node related to the source IP is O(1). In the worst case that all the rules are related to the source IP, we need two comparison for destination IP and rule’s priority to verify the relevance of the rules. The maximum number is O(n).

For a request that took place most recently, we just have constant time O(1) operation to find it within the most recent list and apply it again.

4.2 Space Complexity

In the worst case, we need O(log n) nodes to store data where n is the total number of the rules (proof is available). Each rule needs a unit space to store, in worst case that rules are unique and there is no redundancy or conflict in rule set. Therefore the total space required is O(n+log n) which is O(n).

5 Optimization

Since the traffic between specific IPs happens frequently, we make a limited size of the most recent rules applied to the network traffic, this list is separated from our binary tree . We first check the list when a traffic request comes, if the rules are not on the recent list, we search back to our whole rule sets. This list will be updated after each event. The event is applying a rule to traffic or any changes in rule sets. In reality, as the coming traffic streams constantly, the recent rule list makes the firewall faster because we know that every nodes in Internet does not request to connect to other nodes by checking our whole rules table. Access to this list is very fast, which is in constant time O(1).