Techniques for Anomaly Detection in Network Flows

Iván O. García Humberto Ortiz-Zuazaga

December 22, 2013

Abstract

In terms of IP networks, anomalies are unusual and significant changes in a network's traffic level. At the present, researchers work to find methods that help detect anomalies efficiently. In this report, we explore two methods for detection. First, we see how the subspace method can be used to detect anomalies in the following different types of traffic flows: byte counts, packets counts and IP-flow counts. Next, we see how Benford's law is used to detect anomalies in inter-arrival SYN flow packets. Then, we make an implementation which separates and identify IP numbers that generated the most traffic in a network. Another implementation was made with the use of the Benford's law. These methods detect anomalies and do not offer a description, nor a type of it.

1 Introduction

Anomaly detection is the process of detecting unusual and significant changes in a network's traffic level. Traffic is a collection of network flows which are IP packets with common characteristics passing through an observation point at a given moment [4].

Due to the increasing traffic in world-wide networks, it is not a simple task to process flows because it is resource intensive.

We describe two methods previously reported in the literature: the subspace method, and an implementation of Benford's law. Then we partially implement each of these techniques, and apply them to a collection of flows from the UPR's network.

2 Prior work

In the two papers that we explored [1] and [2], the methods successfully detected anomalies. In the first paper, with the subspace method along with the Principal Component Analysis applied to byte counts, packets counts and IP-flow counts, the following type of anomalies were detected: alpha, dos/ddos,

flash crowd, scan, worm, point to multi-point, outage and ingress-shift. Similar anomalies were detected with the used of Benford's law applied to inter-arrival SYN packets.

2.1 Subspace Method

The subspace method is used for anomaly detection on bytes, packets and IP-flows counts in a network. It works by separating a whole group of traffic in equal intervals of time. Then it uses the Principal Component Analysis(PCA) in each interval of time to reduce the multivariate system in another with fewer variables [3]. Then, it choose the principal components which contains a significant variance. Those components are the anomalous subspace corresponding to anomalies. Finally, PCA is applied in interval of time with each type traffic. The discarded components(no significant variance) are the normal subspace of the original multivariate set [1].

2.2 Benford's Law

The Benford's law is known as the law of anomalous numbers. It states that the frequency of a digit (d) appearing as a leading digit in a collection of numbers, follows this logarithmic relation: $P_d = \log_{10} \frac{(d+1)}{d}$.

If we compare Benford's law with inter-arrival times of SYN packets, then anomaly are treated as significant discrepancies between them. The use of this law allow anomaly detection in real time, it not resource intensive and is not a complicated approach. [2]

3 Methods

3.1 Implementations

In order to have a better idea of how to handle the data used in the implementation of the subspace method, we wrote a C++ program that identifies first 10 IP-numbers that generated the most traffic in a network. The program start by reading a file which contains Ip Numbers with its respective octets. Then, it sums the octets for repeated Ip-numbers. Finally, it orders the list of Ip numbers, placing those with more octets at the final and we obtain the first 10 IP numbers with the most octets.

```
// Programmer: Ivan O. Garcia
// Date       : August 28, 2013
// Purpose: To read a file called "test.out" to extract all sources IP with
// its octets to see what IP address have the most traffic.

#include <iostream>
#include <iomanip>
#include <fstream>
```

```
#include <string>
#include<map>
using namespace std;

int main() {
// Reading 'test.out' file.
        // Creating an input file variable.
         ifstream in ;              // Variable to access the file.
         in.open("test.out") ; //Opening file.

    string srcIP, dstIP ;       // To hold srcIP and dstIP.
        int prot, srcPort, dstPort,// To hold respective data.
            octets, packets ;

        // Skip the header line.
        string line ;
        getline(in, line) ;

    map<string,int> matches ; //Uses 'map' function to match 'srcIP'
                              //with 'oc

 matches[srcIP]= 0 ; // To have something in the 'map'.  tets'.

        // To iterate each file's line.
        while(!in.eof()) {

                // Primming read.
                in >> srcIP >> dstIP >> prot >> srcPort
                 >> dstPort >> octets >> packets ;

        map<string,int>::iterator it ; //To iterate the 'map'.
        it = matches.find(srcIP) ; // if 'srcIP' is in the list: it = address of
                                   // 'srcIP'

      if(srcIP == (it->first)){//new 'srcIP'='srcIP' in 'map': do the following.
            int oct_temp ;  //To hold old 'octets' count.
            oct_temp = (it->second) + octets; // new 'octets' + old 'octets'.
            matches.erase(it) ;  // Erase old 'key' and its value from 'map'.
            matches[srcIP] = oct_temp ; //Places new 'key' and its value.

            }
        else
            matches[srcIP] = octets;//Places new 'key' and its value in the map.
        } // End of 'while' cycle.

map<string,int>::iterator i ; //To iterate the 'map'.
```

```
string arr_IP[matches.size()]; // To hold all 'srcIP' from map.
int    arr_oc[matches.size()]; // To hold all 'octets' from map.

int a = 0 ; // To iterate the arrays.
for (i=matches.begin(); i!=matches.end(); i++) {
       arr_IP[a] = i->first;  // assign 'srcIP' from map to the array.
       arr_oc[a] = i->second; // assign 'octets' from map to the array.
       a++ ;
}

// Using 'Bubble Sort' algorithm to sort the IP addresses by octet quantity.
for(int i=0; i < matches.size() -1 ; i++){
     for(int h=0; (h < (matches.size() - i - 1)) ; h++) {

          if(arr_oc[h] > arr_oc[h+1]) {
             string IP_temp =  arr_IP[h];
             int oc_temp = arr_oc[h];
             // Swap
             arr_IP[h] =  arr_IP[h+1];
             arr_oc[h] = arr_oc[h+1];
             // save temporary variables to the arrays.
             arr_IP[h+1] = IP_temp;
             arr_oc[h+1] = oc_temp; }
          } // End of 'for' cycle.
 } // End of the first 'for' cycle.

// Print the arrays.
for(int i=0; i < matches.size() ; i++)
 {
   std::cout << setw(18) << arr_IP[i] << setw(8) << arr_oc[i] << std::endl ;
 }

// Closing file
        in.close() ;

return 0 ;
} // End of 'main' function.
```

In relation to the Benford's law, we made a Python program which receives a set of flows, calculate the frequency according to the law and we compare it with the original set of flows. It begins with reading a file of flows and save the arrival time between each flow(inter-irrival time).Then, it save the frequency of the numbers 1-9 as leading digits in the inter-arrival times and plot individually each frequency. Then, it substitutes the numbers 1-9 in the Benford's law and also plot each of it individually in a same coordinate system. Then, a comparison

between the two curves was made.

```
import flowtools
from collections import defaultdict
import pylab
import math

def check_flowstarts(flowstarts):
    "check the leading digit distribution for a set of flow start times"
    counts = defaultdict(int)
    denom = len(flowstarts)

    for i in range(1, denom):
        last = flowstarts[i-1]
        cur = flowstarts[i]
        if cur == last:
            continue

        digit = int(("%.3e" %  (cur - last))[0])
        counts[digit] += 1

    digits = range(0,10)
    freq =  [1.0 * counts[d] / denom for d in digits]

    return digits[1:10], freq[1:10]

def rms(freq, expected):
    sum = 0
    for i in range(len(freq)):
        sum += (freq[i] - expected[i])**2
    return math.sqrt(sum)

### main

set = flowtools.FlowSet( "-" ) # Read from stdin

flowstarts = []
#laststart = 0.0
for flow in set:
    #    if laststart != flow.first:
    flowstarts.append(flow.first)
    #    laststart = flow.first

flowstarts.sort()

expected = [ math.log(1.0/d+1.0,10) for d in range(1,10)]
```

```
xs = []
ys = []

for i in range(len(flowstarts) - 1000):
    digits, freq = check_flowstarts(flowstarts[i:i+1000])
    #xs.append(flowstarts[i])
    print flowstarts[i], rms(freq,expected)
```

3.2 Data

To test our programs, we applied them to the analysis of a dataset from the University of Puerto Rico's High Performance Computing facility. The data consists of 683 MB of flow data collected September 21, 2011, with over 38 million flows.

4 Results

Our C++ program shows that only few IP numbers generate the majority of a network's traffic. On the other hand, in the implementation of Benford's law, we graphically showed the differences between Benford's distribution and the original set of flows.This program called check-benford.py takes 16 hours to process one day's worth of flows (683 MB, 38M flows).

5 Future work

In a future, we will explore how we can extend our work by finding a way to implement the Subspace Method and identify which type of anomaly have been detected. Finally, for the Benford's law, we have to manually classify each anomaly.

6 Conclusion

Anomalies represent a threat because it can decrease a network's speed by creating congestion. To help to manage this problem, we first need methods to detect the anomalies to subsequently identify the anomaly and finally correct it. The Subspace Method with Principal Component analysis and the Benford's law, provide statistical models which are easy to implement and more important, they do not consume too many resources.

References

[1] Lakhina Anukool, Mark Crovella, and Christophe Diot. Characterization of network-wide anomalies in traffic flows. citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.92.7738&rep=rep1&type=pdf, 2004.

[2] Laleh Arshadi and Amir Hossein Jahangir. Benford's law behavior of internet traffic. *Journal of Network and Computer Applications*, 2013. doi:10.1016/j.jnca.2013.09.007.

[3] Dong Hyun Jeong, Caroline Ziemkiewicz, William Ribarsky, and Remco Chang. Understanding principal component analysis using a visual analytics tool. http://www.vrissue.com/portfolio/pdf/UKC2009.pdf.

[4] J. Quittek, T. Zseby, and S. Claise, B.and Zander. Requirements for ip flow information export (ipfix). http://tools.ietf.org/html/rfc3917#section-2.1, 2004.