Using Visualization to Improve Analysis of Anomaly Detection in IPv6 Flows

Grace M. Rodríguez Gómez
grace.rodriguez6@upr.edu
University of Puerto Rico – Rio Piedras Campus
Advisor: Humberto Ortiz-Zuazaga
humberto@hpcf.upr.edu
Faculty of Natural Science - Computer Science Department
University of Puerto Rico

Abstract

In order to ensure long-term growth, network operators and companies have been starting to implement IPv6 in their network. However, new individual products that use IPv6 are very exposed to malicious attacks because their bugs and security vulnerabilities haven't been discovered and fixed yet. That's why we have decided to test IPv6 flow data to find new ways to improve security. One method that researchers are exploring to improve web security is anomaly detection in traffic flows. In this research, we are classifying a flow anomaly those packets with an inexplicable amount of data (bytes).

Even though the implementation of IPv6 is relatively new, there can be a lot of flows in just one day. Analyzing all of these flows individually is an almost impossible and very tedious task. Therefore, we have decided to create and use visualization tools that will allow us to picture our data better and analyze them more efficiently. One approach we have taken is to use the fields in IPv6 flows data of source Address, destination Address and Destination Port, as coordinates to display a 3D cube.

1. Introduction

IPv6 was developed by the Internet Engineering Task Force (IETF) to deal with the long-anticipated problem of IPv4 address exhaustion [8]. Currently, there has been seen a growth in IPv6 traffic in the past years. This means that more people are using IPv6 on their personal Internet routers and firewalls. It also means that more hackers are going to try and exploit vulnerabilities in IPv6 with malicious attacks. That's why it's important to search for ways to detect these vulnerabilities and implement protection so users' data can travel safely through the network.

For this reason, we have decided to test IPv6 flow data to find new ways to improve security. We define a flow as a set of IP packets passing an observation point in the network during a certain time interval [8].

We want to concentrate in making new visualizations tools so the process of analyzing flow data can be facilitated. Visualization is necessary to understand abstract data better, especially when there is a big amount of it. We would also like for these tools be available to the general public once they're fully tested.

Moving from our last research, we want to display a 3D cube whose particles are IPv4 and IPv6 traffic flows that are being fed in real time. We will be using the university's network to get test data and a visualization tool that was already developed by Dr. José Ortiz and his research students.

2. Past work

We will parting from our past research of Fall 2015 where we converted IP addresses to coordinates to create a 3-dimensional cube. We want to be able to display flow data that is being fed in real time instead of static data. We will be using a plugin from TOA, a web based NetFlow data network monitoring system (NMS) [2]. The plugin that we will be mostly working with is a 3d cube that has already been implemented that displays flow traffic, similarly to the one done in our own research. However, this cube only displays IPv4 data. Hence we have decided to edit the code so it also takes IPv6 flows as input.

3. Methods

3.1 Getting familiarized with TOA

Toa consists of a collection of scripts that automatically parse NetFlow data, store this information in a database system, and generate interactive line charts for network visualization analytics. The system is pseudo real time, meaning that it continuously updates the interactive charts from NetFlow data that is generated every five minutes. Toa also provides an interface to generate customized charts from the data stored in the database, and plugins that connect the visualization charts with the NetFlow data file for more in depth visualizations and analysis [2].

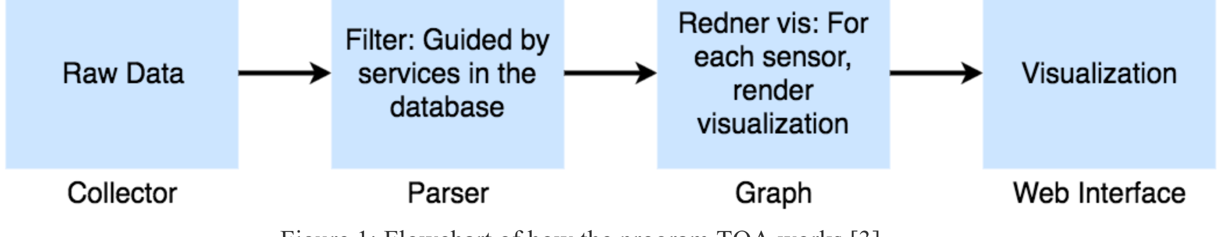
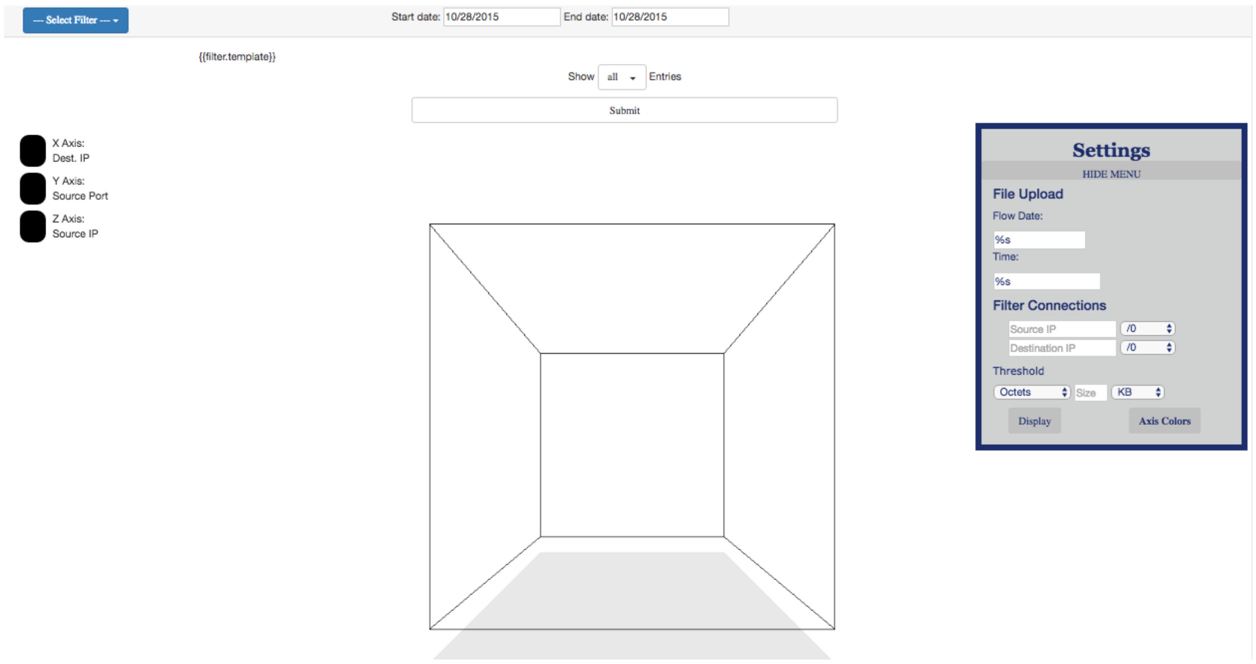

Raw Data
Filter: Guided by services in the database
Redner vis: For each sensor, render visualization
Visualization
Collector
Parser
Graph
Web Interface

Figure 1: Flowchart of how the program TOA works [3]

However, its development has been discontinued for now. For this reason, we have decided to move the cube plug-in to another web API (Application Programming Interface) that was developed by Julio de La Cruz and Dr. José Ortiz. The API is located in the website http://wolverine.ccom.uprrp.edu/. The server that hosts this site captures IPv4 and IPv6 flow data from our university's network.

3.2 Understanding the cube's interface

For those people with knowledge in networks, the cube's interface is pretty straightforward. In Figure 1, we can see at the upper left corner a drop down menu titled as "Select Filter". The options in the dropdown menu are Input Interface, Output Interface, Source IP, Destination IP, Source Port, Destination Port, Bytes and Packets.


Start date: 10/28/2015 End date: 10/28/2015

Show all Entries

X Axis: Dest. IP
Y Axis: Source Port
Z Axis: Source IP

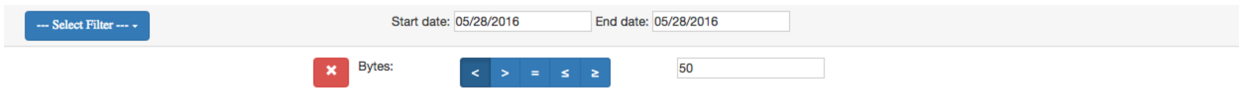
Flow Date: %s
Time: %s

Source IP /0
Destination IP /0

Threshold: Octets Size KB

Figure 2: The 3 Dimensional Cube interface for displaying flow data

With the options in this dropdown menu, we can choose what type of flow data we want to display on the cube. In this research, we're mostly going to use the filter of Bytes, since in our last research we define as an anomaly an inexplicable big amount of bytes within one flow and it's also the simplest way to filter the flows.



Start date: 05/28/2016 End date: 05/28/2016

Bytes: 50

Figure 3: How to filter flow traffic with the amount of bytes

In able to filter the IP flow with bytes, the user writes the desired amount of bytes they want their flow data to have. Then they choose if they want the IP flows to have an amount of bytes bigger, less or equal to the amount they entered with the respected html button.

3.3 The code that makes the cube run

The cube has four main scripts that make it work, which are ctrpan.js, cube.py, index.html and net_tools.py. What can be called the heart of the program would be ctrpan.js. This is the script that sends the flow data to the cube, parses it, and sends it to index.html so it can be displayed as particles. Index.html mostly the front end that displays the cube and the buttons to filter the data. All of the flow data is sent through a json from the servers to the web API. This is done in the function *dspData(request_type)* in the ctrpan.js script.

```
1. function dspData(request_type)
2. {
3.      temp_list = request_type;
4.      for(var i = 0; i < temp_list.length; i ++){
5.      flow_records.push(temp_list[i]);
6.      buildParticles() ;
7.      setInterval(updateParticles,3000) ;
8. }
```

Script 1: Function that receives all the flow data in a json

The line 3 receives all the flow data as a dictionary of dictionaries via a json and saves it the variable temp_list. This variable is then parsed in the for loop in line 4. Meanwhile, the function in line 6, *buildParticles* gets the data by value and displays them in the cube.

3.4 Implementing IPv6

The changes that had to be done to the *ctrpan.js* script in order for the cube to process IPv6 data as well as IPv4 weren't that many. Mostly, we have to make sure that the script could parse to 128-bit address instead of a 32-bit address. This is because an IPv6 address is 128 bits long and has 2^{128} addresses while an IPv4 address is 32 bits long and only offers 2^{32} addresses.

The first change that was made was in *buildParticles* function. The line

ratepos = parseFloat(flow_records[index][2])/Math.pow(2,**32**-dst_prefix);

Was changed to **ratepos =** parseFloat(flow_records[index].dip)/Math.pow(2,**128**-dst_prefix);

This line parses through the variable that is the flow data and divides it 2^{128} - destination prefix.

Another similar change was the line in the same *buildParticles* function ratepos=parseFloat(flow_records[index][1])/Math.pow(2,**32**-src_prefix);
That was changed to

ratepos=parseFloat(flow_records[index].sip)/Math.pow(2,**128**-src_prefix);

Ratepos is going to become the integer of each source and destination address that the cube is going to receive in order to display the data.
Another function that was also modified was the one called *getIpOctet.* This function parses through each IP address and replaces its block divider by white space.

```
function getIpOctet(ip,octet_numbers){
        if (ip.lengt=0||octet_numbers>4||octet_numbers<0)return ""
        ip=ip.split('.'),newIp="",ip_index=0;
        for(i=0;i<octet_numbers;i++){
        newIp+=ip[i]+(ip_index!=octet_numbers-1?".":"");
        ip_index+=1;
        }
return newIp;
}
```

Script 2: How the function was written before making the IPv6 implementations

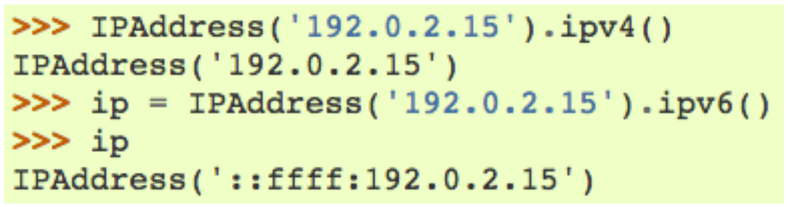
```
function getIpOctet(ip,octet_numbers){
        if (ip.lengt=0||octet_numbers>8||octet_numbers<0)return ""
        ip=ip.split(':'),newIp="",ip_index=0;
        for(i=0;i<octet_numbers;i++){
        newIp+=ip[i]+(ip_index!=octet_numbers-1?":":"");
        ip_index+=1;
}
return newIp;
}
```

Script 3: Function getIpOctets after IPv6 implementations.

In line 2, the 4 was changed to 4 because IPv6 is represented as 8 groups of four hexadecimal digits, meanwhile IPv6 only has 4 groups of four decimal digits. IPv6 addresses are separated by colons, while IPv4 addresses are separated by points. That's the reason why line 3 and 5 where changed.

3.5 Adding Netaddr

While testing, we realized there was an issue regarding displaying IPv4 flows in the cube. IPV6 is not designed to be backward compatible with IPv4. So the easiest way to display IPv4 flows particles alongside IPv6 flows particles was by converting IPv4 addresses to IPv6 addresses. We found out that there is a library in Python that has functions that convert IPv4 to IPv6 called netaddr. Netaddr is a Python library for representing and manipulating addresses. It provides support for layer 3 and 2 addresses [7].



```
>>> IPAddress('192.0.2.15').ipv4()
IPAddress('192.0.2.15')
>>> ip = IPAddress('192.0.2.15').ipv6()
>>> ip
IPAddress('::ffff:192.0.2.15')
```

Figure 4: Example on how to convert an IPv4 address to an IPv6 address using Netaddr

The IPAddress class is used to identify individual addresses. If the ipv6() function returns an IPv4 address, a numerically equivalent version of 6 IPAddress will be returned [7].

The functions to convert from IPv4 to IPv6 where added in the script *webflow.py.*

```
def toJsonInt6(silkDic):
 jsonDic = []
 for item in silkDic:
        rJson = {}
        sip = IPAddress(str(item.sip)).ipv6()
        dip = IPAddress(str(item.dip)).ipv6()

        rJson['sip'] = int(sip)
        rJson['protocol'] = item.protocol
        rJson['timeout_killed'] = item.timeout_killed
        rJson['dport'] = int(item.dport)
```

```
            rJson['output'] = item.output
            rJson['packets'] = int(item.packets)
            rJson['bytes'] = int(item.bytes)
            rJson['tcpflags'] = str(item.tcpflags)
            rJson['uniform_packets'] = item.uniform_packets
            rJson['application'] = item.application
            rJson['sensor_id'] = item.sensor_id
            rJson['timeout_started'] = item.timeout_started
            rJson['classtype_id'] = item.classtype_id
            rJson['stime'] = str(item.stime)
            rJson['nhip'] = str(item.nhip)
            rJson['duration'] = str(item.duration)
            rJson['input'] = item.input
            rJson['sport'] = int(item.sport)
            rJson['dip'] = int(dip)
            rJson['finnoack'] = item.finnoack
            jsonDic.append(rJson)

    jsonDic = json.dumps({'flows': jsonDic})
    return jsonDic

def toJsonInt4(silkDic):
    jsonDic = []
    for item in silkDic:
            rJson = {}

            sip = IPAddress(str(item.sip)).ipv4()
            dip = IPAddress(str(item.dip)).ipv4()

            rJson['sip'] = int(sip)
            rJson['protocol'] = item.protocol
            rJson['timeout_killed'] = item.timeout_killed
            rJson['dport'] = int(item.dport)
            rJson['output'] = item.output
            rJson['packets'] = int(item.packets)
            rJson['bytes'] = int(item.bytes)
            rJson['tcpflags'] = str(item.tcpflags)
            rJson['uniform_packets'] = item.uniform_packets
            rJson['application'] = item.application
            rJson['sensor_id'] = item.sensor_id
            rJson['timeout_started'] = item.timeout_started
            rJson['classtype_id'] = item.classtype_id
            rJson['stime'] = str(item.stime)
            rJson['nhip'] = str(item.nhip)
            rJson['duration'] = str(item.duration)
            rJson['input'] = item.input
            rJson['sport'] = int(item.sport)
            rJson['dip'] = int(dip)
            rJson['finnoack'] = item.finnoack
            jsonDic.append(rJson)

    jsonDic = json.dumps({'flows': jsonDic})
    return jsonDic
```
Script 4: Function that use the Python library to convert IPv4 to IPv6 and viceversa

If the function *toJsonInt4* is invoked, it will return all of the IP in IPv4 format and as an int. If the *toJsonInt6* is invoked, then it will return all the IP in IPv6 format and as an int.

4. Results

Testing out static IPv4 flow data in the cube once it was moved to the web API was quite simple and fast. When the cube interface was working in the website, we tested it out with static ip flow data.

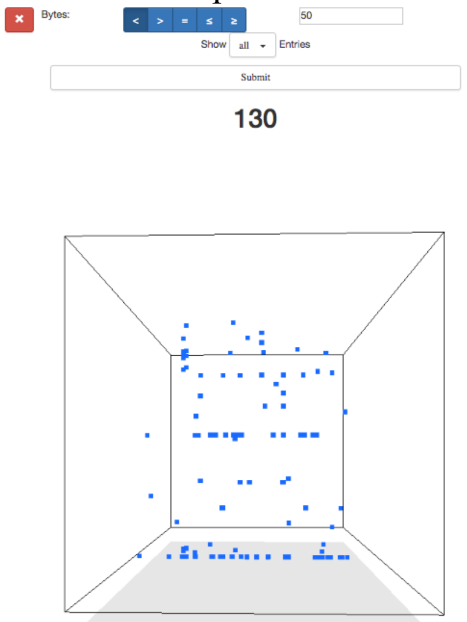


Figure 5: Cube display of IP flow particles made by static data

However, we did run into many obstacles trying to display IPv6 particles. After doing all the modifications that were mentioned in methods, we were able to display IPv6 particles in the cube.

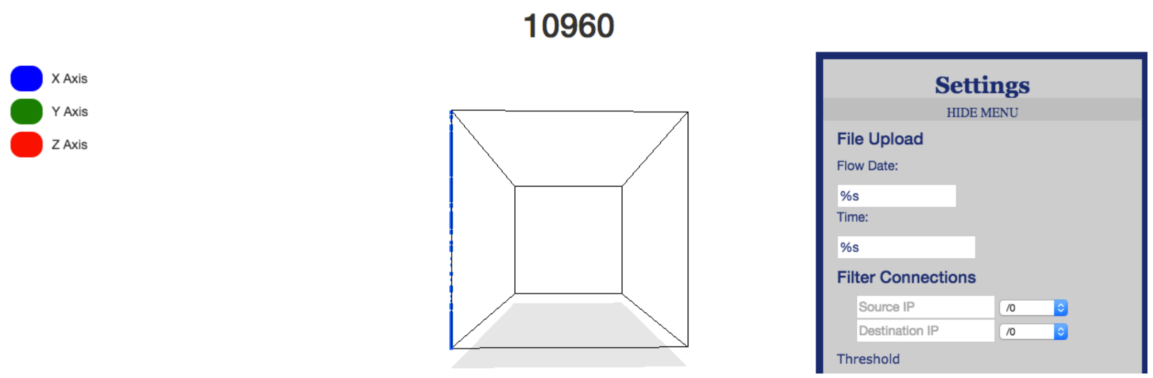


Figure 6: First result displaying IPv6 flow data.

As we can see in Figure 6, the results do not look like the results from the IPv4 cube. The flow particles are all been displayed in one line instead of floating in the center of the cube. We believe this has to do with the fact that IPv6 addresses are so much longer than IPv4 addresses. For IPv4 addresses the maximum number the y-axis (source IP) and z-axis (source IP) are going to have is 2^{32}. Meanwhile, in the case of displaying IPv6 addresses, the maximum number is 2^{128}, which is a lot more. This also has to do with the fact that since we're using flow data from our university's network, the first 64 bits are always going to be the same. Since most of the IP addresses have such a much smaller range compared to the total range of the axis, then all of the particles appear in a straight line. The best approach to fix this would be to make the cube ignore the first amount of bits that define the networks address. For example, an IP address from our university has 64 fixed bits. In this case, we would choose in Filter Connections for range of the y and z axis to start in 2^{64} and end in 2^{128}**.** That way the range of these axes is a lot less and we are supposed to see the particles floating all over the cube.
This approach was taken in the *index.html* script and the *ctrpan.js.* In the *ctrpan.js* script, the line
(src_pre%8!=0||src_pre<0||src_pre>**32**||dst_pre%8!=0||dst_pre<0||dst_pre>**32**) return 0 ; from the function ValidData was changed to if (src_pre%8!=0||src_pre<0||src_pre>**128**||dst_pre%8!=0||dst_pre<0||dst_pre>**128**)return 0 ;
This way, all of the numbers divisible by 8 from 32 to 128 are included.

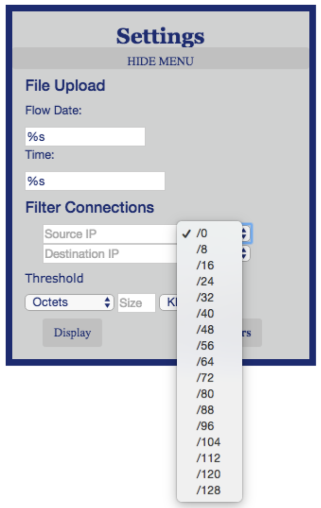

Flow Date: %s
Time: %s

Figure 7: Dropdown menu of all possible filters from 0 to 128

However, when the cube was tested again we got the same results as figure 6. This means the error might be something deeper and we need to study the cube's code more thoroughly to fix it.

5. Conclusion

We were able to move the cube's scripts to a working web API for testing and deploying. We were also able to display IPv4 flow particles in the cube in the web API. We were also able to display IPv6 flow particles, though not exactly as desired.
For the other part, the cube is a good visualization tool in order to monitor the IP traffic in your network and for network forensics. If there is a malicious attack in a server, we can use the cube to see in which date we got an inexplicable amount of data, and try to trace it to see from which IP address it originated.
There were of course some obstacles that are yet to be fixed, the most prominent one being making the cube display the IPv6 flow particles correctly. We're not full sure yet what is causing that the particles are still being printed in a straight line. One assumption could be that even though the change in the /bit was being implemented, the command is not being sent correctly from index.html to ctrpan.js and the cube is ignoring it.

6. Future Work

As mention in the conclusion, there are some bugs that need to be fixed, the main one being the display of the IPv6 particles in the cube. Once that is correctly developed, we would like to make this visualization tool open to anybody that is interested in using it. Right now, the cube only works if

you're connected to a network inside the UPRRP. Making it accessible means that more people can use it and more people can contribute to it new ideas and better implementations.
As we learned, the data is easier to understand if its being displayed in visualization tools such as graphs. We would like to developed more visualization tools that help manage and monitor networks.

7. Acknowledgements

First of all, we would like to thank Dr. José Ortiz for helping us by providing the code, the explanation of how things work and the suggestions to move the cube to the webserver and make the implementation for it to display IPv6 flows. Special thanks to Julio de la Cruz for helping with the implementation and for lending his web API to use it as base to display the cube's interface.

We also want to thank our research advisor, Dr. Humberto Ortiz-Zuazaga, for pointing us in the direction when we were lost with the topic, especially when learning SiLK tools and helping us answer out doubts in the implementations.

This work is supported by the scholarship Academics and Training for the Advancement of Cybersecurity Knowledge in Puerto Rico (ATACK-PR) supported by the National Science Foundation under Grant No. DUE-1438838.

8. References

[1] IPv4 vs. IPv6 Traffic Statistics on Routers. *Blog Webernetznet*. N.p., 23 Oct. 2014. Web. 10 Apr. 2016. <http://blog.webernetz.net/2014/10/23/ipv4-vs-ipv6-traffic-statistics-on-routers/>.

[2] Ortiz, Jose, Humberto Ortiz-Zuazaga, Eric Santos, Albert Maldonado, and Jhensen Grullon. TOA. *GitHub*. N.p., 24 Oct. 2014. Web. 2 Feb. 2016. <https://github.com/cslab-uprrp/toa>.

[3] Ortiz, Jose. Toa: A Web-Based NetFlow Data Network Monitoring System. *Carnegie Mellon University - Software Engineering Institute*. Carnegie Mellon University, Jan. 2015. Web. 4 Feb. 2016. <http://resources.sei.cmu.edu/library/asset-view.cfm?assetID=431211>.

[4] Roberts, Phil. Internet Society. *Internet Society Blog*. N.p., 02 Dec. 2014. Web. 29 Mar. 2016. <http://www.internetsociety.org/blog/tech-matters/2014/12/google-ipv6-traffic-passes-5-ipv6-internet-growing-faster-ipv4?gclid=Cj0KEQjwid63BRCswIGqyOubtrUBEiQAvTol0d_jDpSOVuvrxYBfkAJPKwVS_Q2Z1erlpb32qgmRI9waAoiz8P8HAQ>.

[5] Rouse, Margaret. What Is Network Forensics? - Definition from WhatIs.com.

SearchSecurity. N.p., Sept. 2005. Web. 29 May 2016. <http://searchsecurity.techtarget.com/definition/network-forensics>.

[6] IPv6. *Wikipedia*. Wikimedia Foundation, n.d. Web. 1 May 2016. <https://en.wikipedia.org/wiki/IPv6>.

[7] Netaddr 0.7.18 : Python Package Index. *Netaddr 0.7.18 : Python Package Index*. N.p., n.d. Web. 29 May 2016. <https://pypi.python.org/pypi/netaddr>.

[8] IPv6 Tutorial. *Tutorialspoint*. N.p., n.d. Web. 20 Mar. 2015. <http://www.tutorialspoint.com//ipv6/index.htm>.

MLA formatting by BibMe.org.