Network Visualization

Jun “Albert” Kim, Calvin Ong, Zhuo Hao “Allen” Zhang

Network Visualization Front-End

To build the front end of the network analysis website, the bootstrap framework was used. The basic version of Bootstrap was used (i.e. not with Source or Less). The advantage of using Bootstrap was that no media queries were necessary and all the graphs and navigation bar scaled easily. Any future graphs or analyses that needed to be added to the website could be also done so easily without needing to modify the entirety of the code. Simply adding another container, column, row and graph is all that is needed.

```
<div class=”container”>
    <div class=”row”>
        <div class=”col-lg-12 col-xs-12”>
            <div id="analysis#"></div>
        </div>
    </div>
</div>
```

For the navigation bar, simply adding another link and changing the CSS of the body to allow more space between the body and navigation bar is all that is needed.

```
<li class="link" id="analysis#link"><a href="analysis#/analysis#.html">Analysis #</a></li>

.body {
   margin-top: …;
}
```

To create the graphs, some back-end code was needed to properly extract the needed data from the graph and generate the data as json objects. Once that was done, Plotly was used to generate the graph to put in the container. To use Plotly, the script had to be called.

```
<script src="https://cdn.plot.ly/plotly-latest.min.js"></script>
```

In a separate javascript file, four functions were created:

loadJSON(file, callback)

returnLayout(title, x_axis_title, x_axis_type, y_axis_title, y_axis_type, hovermode)

returnLayout2(title)

load(jsonFile,divID,layout)

The returnLayout and returnLayout2 functions take the string values to create the axes and the location of the legend for each graph. returnLayout is for x-y graphs and returnLayout2 is for

non-x-y graphs like a pie chart. An object is created and the object retains these values. This object is passed onto the load function. The load function calls the json file:

loadJSON(file, callback)

and parses through the values. Each value is passed into an object. Since the values are already cleanly generated through the back-end code, it is not necessary to do the parsing again. The Plotly function:

Plotly.newPlot(divID, actual_JSON,layout)

is used to generate the graph.

The div which the Plotly will place the graph in will need to have an id for which the Plotly function takes in as one of its parameter. It should be noted that the Plotly function will generate two nested divs within the div specified.

```
<div id=“analysis#”>
    <div>
        <div>
        </div>
    </div>
</div>
```

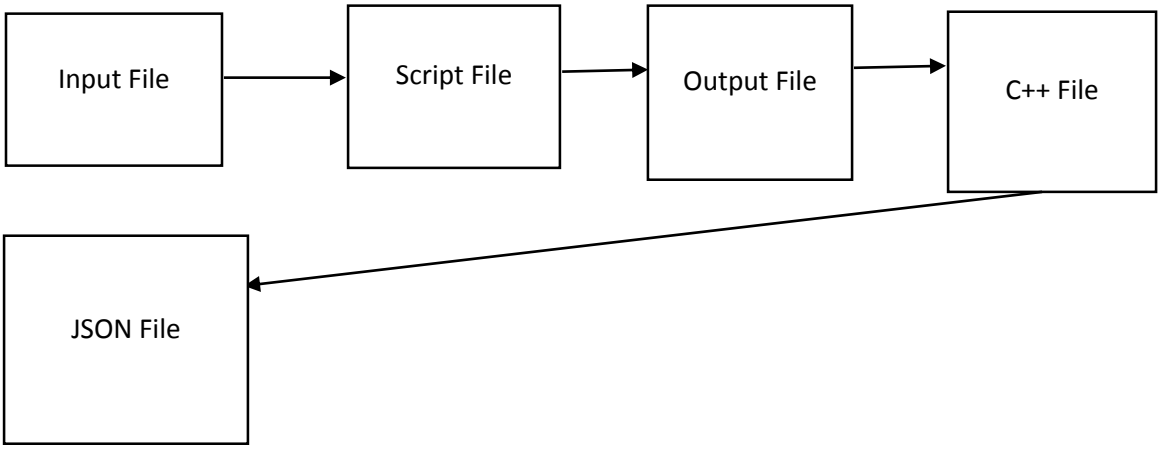
All that is necessary is to write in the html file:

```
<script src="../scripts/plotFunctions.js"></script>
<script>
    var layout = returnLayout(…);
    load("../json/necessaryFile.json","analysis#",layout);
</script>
```

necessaryFile.json is the json file containing the necessary data. The above code is written after all the scripts.

Network Visualization Back-End

To create the back end of the network visualization website, the data had to be organized and the necessary data for each analysis extracted. For example, for Analysis 1, only the protocol data needed to be read. It is not necessary to read the sample data 6 times for each analysis. Once the data is read and processed, the 6 JSON files to be processed by the front-end code is generated.



The input file is located in the IO folder. The input file is the sample data in its original format. When running the script to get the packet data, this input file would be overwritten to get the new data each time.

The bash script file sanitizes the data by replacing the empty columns with 0 and deleting the headers. This includes getting rid of the string: *['test/t1.py','captures/univ1_pt1.pcap']*

```
awk -F"\t" -v OFS="\t" '{
    for (i=1;i<10;i++) {
      if (i == 9){
        $i = ""
      }
     if ($i == " " ) {
       $i="0"}
    }
    print $0
}' ./IO/output > ./IO/output.new && mv ./IO/output.new ./IO/output
```

The new data is written to the output file, which is in the same IO folder as the input file.

In the C++ file, an array of all possible string values for different protocols are created for Analysis 1. The a.out file is generated by the G++ compile command (for linux machines).

void seq()

This function is not used.

vector<vector<string>> vectorProcess()

This function takes no parameters are returns a vector of type string vector. 9 vectors are created for the 9 columns in the sample data- source ip, destination ip, protocol, source port, destination port, timestamp, packet size, flag and sequence number. These vectors of type string are put into a vector called *allcont*, which is what this function returns.

vector<vector<string>>allcont = {vector<string>sourceIp, vector<string>destIp, … }

Each string separated by a whitespace character in each column are read and put in each vector accordingly. Since the data was sanitized by the bash script file, empty columns are not an issue.

getHeatMap(vector<vector<string>>& allcont)

This function can be used for a heat map.

void getFrequencySize(vector<vector<string>>& allcont)

void getFrequencyProtocol(vector<vector<string>>& allcont)

void getICMPOthers(vector<vector<string>>& allcont)

void getResetVsTime(vector<vector<string>>& allcont)

void getFinVsTime(vector<vector<string>>& allcont)

void getSynVsTime(vector<vector<string>>& allcont)

These six functions take *vector<vector<string>>& allcont* as the parameter and call the needed vectors for their respective analyses. For example, getFrequencySize (Analysis 3) calls *allcont[6]* since that is the packet size vector in the allcont vector, which is needed for this analysis. This call is read into a temporary vector within the scope of the function.

An unordered map - *unordered_map<string, int> frequency* is used to count each unique string. So for example in getFrequencySize, an x-y chart is needed, so the unordered map counts the number of times each unique string (the key) occurs and assigns and integer value to it. The implementation varies for the last 3 functions, but the basic idea is the same. Below is the implementation for getFrequencySize.

```
for(int i = 0; i<psize.size()-1; i++){
    if (freqeuncy.count(psize[i])){
       freqeuncy[psize[i]] +=1;
    }
    else if (!freqeuncy.count(psize[i])){
```

```
        freqeuncy[psize[i]] =1;
    }
}
```

Once the data points are created and written into the unordered map, it is written to the json file in the networkAnalysis folder. The json files are created there. As with the data points, the implementation to create the json files are dependent upon each analysis.

The data in the JSON files are in the following format for x-y graphs:
[{"x":["xDataPoint1, xDataPoint2,…

],"y":[“yDataPoint1, yDataPoint2,…], "type" : "bar"}

Note that in C++, the backslash is the escape character necessary for the double quotations.

The *“type” : “bar”* is for Plotly to denote that it is a bar graph. Only getICMPOthers uses *"type" : "pie"*.

For the last 3 functions, if the bit is present in the data point (i.e. 1) and if it is not unique, then the counter is incremented by 1. If it is unique, then the counter is reset and the y value (second integer value) in the unordered map is updated with the final count. Each time is incremented by 1. Note that each timestamp is unique, but only the *diff – ntime* checks only the first couple of decimal places.

For example, if the timestamp was 263.502516985, and the *diff* counter was 264 the difference (*diff – ntime*) would be positive, and it would mean the difference in time between the two timestamps is negligible enough to exclude this data point as unique. However, if the difference was negative, it would mean the data point is unique since we have already checked all the timestamps from where we previously left off (the *diff*) and the current time stamp. Note that the diff will have the same magnitude as the current time stamp (i.e. if the time stamp was around 263, the diff would never be 150). The *diff* is a counter of the timestamp, alongside the counter of uniqueness (*sum*).

string DecimalToBinaryString(int dec)

This function is sued for Analysis 4 (getResetVsTime), 5 (getFinVsTime) and 6 (getSynVsTime). It takes a integer value (the flag integer) and converts it to a 6-bit binary value. This is so the reset bit, fin bit and syn bit can be obtained for the respective analyses.

Network Visualization Analysis

Protocol vs. Frequency

From the sample data, there were 12 protocols found:

1. IGMP - Internet Group Management Protocol
2. EGP - Exterior Gateway Protocol
3. OSPFIGP - Open Shortest Path First Interior Gateway Protocol
4. ICMP - Internet Control Message Protocol
5. UDP - User Datagram Protocol
6. TCP - Transmission Control Protocol
7. PIM - Protocol Independent Multicast
8. ESP - Encapsulating Security Payload
9. VRRP - Virtual Router Redundancy Protocol
10. IGP - Interior gateway protocol
11. GRE - Generic Routing Encapsulation
12. SCPS - Space Communications Protocol Specifications

Any rows with any empty field in the protocol column was considered to have an unassigned protocol network. From the data gathered, the TCP protocol was the most prevalent at the time the data was captured, with nearly half a million exchanges using that protocol (666,096 exactly).

ICMP vs. Others

Based on the sample data, approximately 4.38% of the used protocols were ICMP. ICMP is "used by routers, intermediary devices, or hosts to communicate updates or error information to other routers, intermediary devices, or hosts" (https://support.microsoft.com/en-us/kb/170292). Although this seems like a low percentage, from the Protocol vs. Frequency analysis, it is the fourth most used protocol in the sample data.

Size vs. Frequency

Based on the sample data, the largest packet size transferred is 1500 bytes. The most common packet size is 40 bytes, with 178,207 exchanges using this packet size. The second most common is the largest packet size, with 153,290 exchanges using this packet size.

Reset vs. Time

A reset is the third most significant bit in a flag that determines whether a connection is reset. Based on the sample data, the number of reset connections was greatest at time 204.2572, with 1150 reset connections.

New Connections vs. Time

A new connection is identified by the SYN bit in the flag (the fourth most significant bit). Based on the sample data, the number of new connections was greatest at time 297.1162, with 493 new connections.

Closed Connections vs. Time

A closed connection is identified by the FIN bit in the flag (the fifth most significant bit). Based on the sample data, the number of closed connections was greatest at time 249.4205, with 1279 closed connections.