**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.

JSON File

C++ File

Output File

Script File

Input File

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.