Graph-Based Analysis for IoT Devices with Manufacturer Usage Descriptions - An ScM Research Project

Samuel McKinney

May 15, 2019

Abstract

This project explores how graph embedding representations of network traffic can be leveraged with Manufacturer Usage Descriptions (MUD) to improve analysis and security for Internet of Things devices. To expand on previous approaches, this project builds graph embeddings from NetFlow-style network traffic statistics readily available for enterprise networks and compares them using graph kernel-based algorithms and finds promising results for device identification and classification.

1 Introduction

Today, much of the connected world involves devices simpler than computers or smartphones, but that have access to local networks and the Internet. These Internet of Things (IoT) devices are very useful, but have brought with them a series of challenges that the industry continues to grapple with. Namely, security solutions for IoT devices tend to fall short of most modern standards for computers or phones – due to their wide proliferation and simple functionality, many devices are susceptible to malware that take advantage of their connectivity to carry out cyber-attacks[1] . To counter this trend, the security community has begun to reason about and develop tools for handling unsecured devices that are already out in the world, and recommend better practices for manufacturers to follow as they release new products.

To better manage unsecured devices, the IEFT has proposed a framework called MUD or Manufacturer Usage Description that describes appropriate behavior for IoT devices[2]. These act as registered safelists that the IEFT proposes that device Manufacturers distribute publicly. Comparing these profiles with current device behavior would allow network supervisors or automated systems to identify and analyze the traffic from a given device.

For non-IoT devices, graph based methods of network traffic analysis have emerged[3] and found success in identifying malicious devices and otherwise managing devices in an area network. Graph structures make sense for network traffic because devices can be well represented as nodes, and connections between devices as edges. These usually take the form of blocklists where the intrusion detection task attempts to isolate devices that contact malicious end-hosts. IoT devices are often used as compromised instruments in larger volumetric attacks on external webservers for the purpose of Denial of Service[4]. The reason these attacks may be hard to detect is that the victim websites are usually not malicious and so do not trigger any sort of blocklist style logic systems for intrusion detection. Given the intended limited network functionality of IoT devices in particular, it may be possible to detect anomalous network behavior that deviates from the devices intended

This project extends the safelist-like logic of MUD policies to explore a graph-based approach for specifically analyzing the network traffic of IoT devices. The main work of this project processes raw PCAP

data using a modified version of the University of New South Wales team's tool MUDgee[5] to produce a usable "MUD" profile, that is assumed to be ground truth for a device. Then, Netflow data collected from Brown University's Guest WiFi network is filtered for IoT traffic, which then individual devices are selected from and used to perform graph similarity and classification tasks on. Moderately promising results were found for using graph-based comparison methods to compute similarity between devices known to be the same model.

Different embeddings were explored, where we chose to include or exclude rule and flow features from the data. A strong motivation emerged to explore the use of feature and label based graph analysis measures where we used the graph structure to store data about devices behavior with the abstractions of nodes and edges. At a high level, we chose nodes to represent individual end points, (devices, web servers, local destinations) and edges to represent the properties of their connection (IP protocol, volume of packets, byte counts). While the set of features we ended up using is by no means complete, we hope to provide the beginnings of a framework to process enterprise network data and Manufacturer Usage descriptions as graphs.

Once embedded into graphs, this project explored various Graph Kernels for computing the similarity of a device's proposed safelist (MUD), and its actual behavior on an enterprise network. Graph Kernels are essentially internal products of graphs that translate their structure and content into a vectorized form that can be easily compared with other graphs. A device whose traffic graph significantly deviates from its MUD graph may need to be monitored more closely or quarantined to mitigate cyberattacks going to or coming from the local network.

The eventual goal of this work is to provide a suggested framework for the security community to represent network traffic as a series of graphs and their subgraphs, and use this structure as a means of analyzing network behavior to identify specific devices, classify flows in the network, and detect malicious behavior from IoT devices. This approach makes sense specifically for IoT devices because their behavior is often limited to a predictable set of rules, rather than a computer or phone that is expected to be able to make web requests to any site. For these devices whose functionality is intended to be more limited, we expect to have an easier time identifying the type of device, and whether or not it is playing nice, based on the proposed graph-theory analysis on its network behavior.

Code for this project is open source and can be found on the project's Gitlab page[6] (you must request access). The tools included in this repository are the main focus of this project and do the data processing and graph kernel analysis described above.

The main workflow for our approach was as follows (See Figure 1 for a more complete workflow):

1. Obtain MUD policy for a device
2. Obtain Netflow Statistics from Enterprise Network Administration (Brown Computing and Information Services)
3. Generate Active Traffic Graph from Netflow Statistics
4. Generate corresponding MUD graph embedding, adapting to specific characteristics of enterprise network environment
5. Compute Similarity score between two graphs, indicating how likely it is that the devices are the same

1.1 Related Work

Supporting this work by providing Netflow data from Brown's guest Wifi network is Nicholas DeMarinis research in Rodrigo Fonseca's systems research group at Brown University. Their data pipeline generated IoT traffic statistics from aggregated network flows on the Brown Guest Wifi network and provided us with those Netflow tables to work with. Their work allows us to filter all of the flows found on the Brown Guest Network by time/date, identify IoT traffic, and pick out key information about the flows these devices participate in. This was key to our work because we used these Enterprise Network style statistics as the basis for what a real-world implementation of IoT Graph Analysis would begin with. Note that this

data is at the flow level, not the packet level - it would be unfeasible and ill-advised for a large enterprise network to capture all traffic at the packet level, but many tools for network intrusion detection rely on packet level analysis.

IoT traffic analysis is being done by several research groups. Most closely related to this work, University of New South Wales in Australia developed MUDgee[5] to develop MUD profiles and have work on identifying IoT Traffic in complex Smart Home environments [7]. Their tool and related work analyzes IoT traffic from their lab and accurately categorizes the devices based on statistics from their traffic. Their push for the adoption of MUD is the basis of this project - we run under the same assumption as they did - that MUD will be more widely adopted by the industry to be used as a standard for IoT device traffic monitoring. We used their public IoT traffic capture datasets to generate MUDs for devices that we did not have running in our lab. Their analysis of IoT devices looks at the content of the packet traces from lab-run devices, and proceeds to use state-of-the-art machine learning techniques to classify the devices. Our graph-based approach may have promise for this identification workflow as more deep learning techniques using graph structures emerges[8].

On the topic of graph embeddings is X. Xu et al's work on using Graph Embeddings with deep learning for similarity detection of binary code[9]. This work explores how control flow graphs can be used to determine how similar two binary files are to one another for the purpose of malware detection. Their work goes further into the domain of deep learning, using newer machine learning graph methods to detect malicious code in the graph embeddings they form from binary executables. Relating to this project is the intuition of transferring data in the euclidean space into the graph space for the purpose of machine learning-style classification.

Florian Mansmann et al[3] explore graph based methods for analyzing network traffic for intrusion detection, but with a greater focus of personal computers and mobile devices. Their work is important to the field by introducing force-directed algorithms for visualization that we took inspiration from here. These algorithms allow for a semantically meaningful representation of the graph to be studied by hand, using data from the node attributes to space the nodes in the graph. Unlike our project, this work does not involve any notion of pre-existing safelists, but rather operates in the opposite direction trying to detect when devices in the network form connections with dangerous hosts that are part of known blocklists. This approach is very effective, but in this project we seek to identify malicious traffic that is not necessarily connecting to dangerous websites, but rather volumetrically significant requests benign victim websites.

2 Approach

This source code for this project uses a modified version UNSW's MUDgee tool[5] that analyzes PCAP data from an IoT lab to create a safelist of flows for a device based on assumed-to-be benign traffic. The modifications we made to MUDgee cause the MUD rules to include volume statistics about the flows found and to name the resulting fields in a manner more similar to Netflow. Then we identify a single device from the Brown Guest Netflow data and use networkx to build and visualize graph embeddings from these sources. Then, using an open source python package, GraKeL, we compute graph kernels on the embeddings created from MUD graphs and active traffic graphs to produce similarity scores.

2.1 Abstract Representation

Let us define a graph embedding to be a set of vertices and edges that represent endpoints and their connections in a local network. Graphs are formed from two places: 1. The ground truth graph for a device is assumed to be the graph formed from it's safelist of rules built from its MUD policy. From here on, we will refer to these Ground Truth graphs as MG_{device} The second kind of graph we have decided to name Active Traffic Graph, AG_{device}. An Active Traffic Graph for a device is the graph formed from netflow statistics from a network with known IoT devices on it. With these two sets of graphs, we chose to encode them using the popular networkx package

in python for graph visualization. In this project, we wanted to store information about the connection on the edges, and information about the hosts on the nodes so this representation made sense. Networkx represents them as node and edge "attributes" which act more like labels than vector features. One of our challenges for the networkx comparisons was converting our feature set into a coherent label that can be compared efficiently in a Graph Kernel.

One key feature of our Netflow statistic to graph embedding pipeline is the ability to choose the granularity of IP address to consider as the same node in the graph embedding. When designing our pipeline, we encountered the challenge of parsing the DNS resolution of hostnames that the devices make connections to. MUDgee includes DNS hostnames in it's rules created by processing PCAP traces from devices - however, in the Netflow data collected from Brown University's Guest network (and by proxy, the enterprise traffic representation we base this project on) does not have DNS information, but rather the DNS resolution (IP Addresses) for each flow in the list. To manage the load-balancing feature of DNS, which is that one DNS hostname may map to several IP addresses depending on the location and time of contact, we decided to include a subnet mask for the graph embedding. When building a graph for a device, one may choose whether to differentiate nodes by /32 (full address), /24, or /16 in order to account for the load-balancing present in DNS resolution for web servers that IoT devices contact. This is especially pertinent for the cloud services that many of these devices contact, more often than not, these services are not static mappings from DNS hostname to IP address, however, data center addresses tend to fall within the same subnet, if a large enough mask is chosen.

2.2 Lab Setup and Datasets

Enterprise network datasets like the netflow statistics obtained from Brown Guest do not give information about packet-level statistics. Most approaches to IoT security based on safelists act on this packet-level domain, doing analysis on packets going through the network to determine if the packet should be allowed to go through or not. For this project, we try to analyze IoT traffic, but on the flow level, matching graphs of flows to the rules that the devices on the network are supposed to follow. We obtained high-level netflow statistics from Brown's Guest Wifi network through its CIS department. Part of Nick DeMarinis systems and networking research is focused on this data collection and aggregation, paving the way for us to use this data for making graphs of devices in the network.

2.3 Challenges with Enterprise Network Statistics

A major component of this project was working under the assumption that enterprise network administrators do not have access to packet level data for their network. Brown University's Communications and Information Services that provided the Netflow data that we work with in this project confirm this assumption to be accurate at the scale of a mid-sized University's public Wifi service. After the aggegation and data-processing done by Nicholas Demarinis for his research in the systems group at Brown, the fields we had from this Netflow were time processed, sampling rate , seq number, time flow, bytes, packets, eth type, ip version, ip proto, ip tos, icmp type, icmp code, if src, if dst, device field, class, ip device, ip other, tp device, tp other, dns hostname, dns client id, dns last update, dns id state, device class, device type. Conversely, MUD often has unresolved DNS hostnames that indicate the namespace of a website that a device may contact. To manage this, at the time of forming G_{MUD}, we make a call to gethostbyname() in the pythons socket library that resolves the name to an IP address. By combining this with subnet masking, we attempt to recognize load-balancing done by the cloud services that are often the hosts being contacted by IoT devices.

2.4 Graph Comparison Methods

Once our graphs have been built and populated with the relevant features, it is necessary to develop methods for comparing two graphs, producing some notion of a similarity score, or confidence interval, that

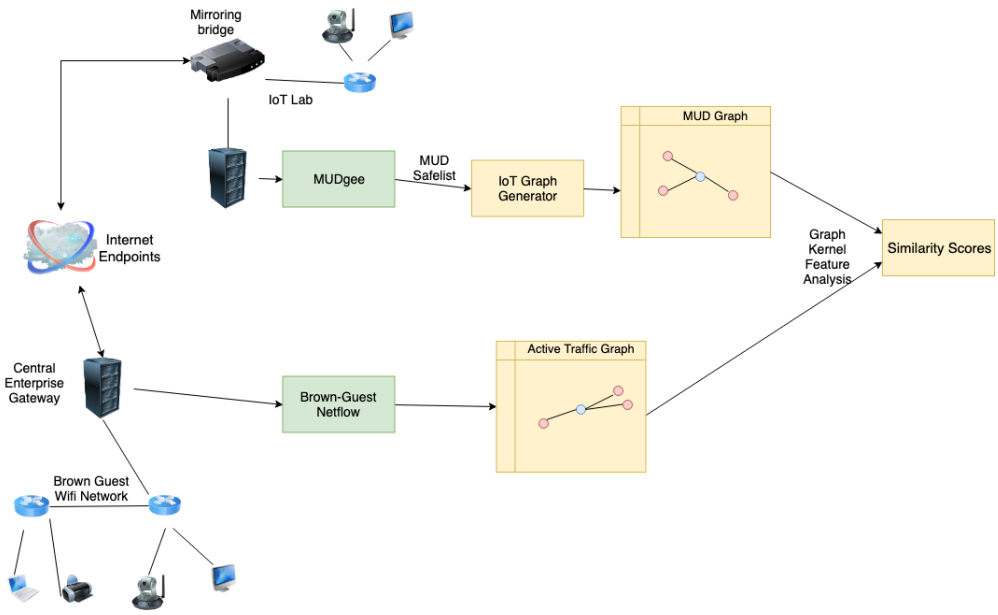

Figure 1: IoT Graph Analysis Workflow for this Project

they belong to the same device. This proved to be a non-trivial task, as our graph representations contain import feature information on both the nodes and the edges. The methods we explored include traditional graph theory methods of structural isomorphism and graph kernel techniques. Here we explain a bit of background on each technique:

Traditional graph comparison methods rely on notions of graph isomorphism [10], which analyze two graphs to try and form a bijection between the nodes of one graph and the nodes of the other such that the adjacency map of the graph is preserved. This is a satisfying comparison method when nodes encode nothing other than a label because it seeks to relabel graphs to make them the most structurally similar to the other graph. To implement this measure, we used networkx's built in graph_edit_distance() function that iteratively determines how many node or edge additions, deletions, or modifications would be necessary to produce a graph that is structurally isomorphic to the other graph. The major downside to this approach is that it is not known to be computable in polynomial time [10]. We found that trying to compute it with graphs larger than 50 nodes took far too long to be a feasible similarity measure. The other downside is that it does not account for node labels, which in our case contains the important information about IP address and protocol that identify flows.

The second area of methods of graph comparison that we attempt make use of Graph Kernels. In the domain of graph theory intersecting with machine learning, Graph Kernels are essentially vector representations of graphs that can be used to compute their similarity[11]. They are computed by several different processes of generating internal products of the graph structure and data. That is, given a graph and the features/labels encoded on its nodes/edges, output an internal product that is comparable with other graphs. Random Walk, Shortest Path, Weisfeiler-Lehman, Graphlet Sampling, SVM Theta, Neighborhood Hash, and Pyramid Matching.

Taking a closer look at two of the kernels we computed, Shortest Path Kernels are an extension of random walk kernels which are computed by counting the number of common walks between two graphs.

A walk is a sequence of nodes that allows repetitions. Shortest Path Kernels do this but on the Floyd transformation of a graph, only considering walks of length 1 to compute all of the shortest paths between pairs of nodes in the two graphs[12]. This measure also does not consider data stored as features on the graphs, but may hold promise for determining if graph structure alone can identifying devices. On the other hand, Weisfeiler-Lehman Graph Kernels are computed using the labels encoded on the graph's nodes[13]. The algorithm is as follows: For each node, concatenate its neighbors' labels with its own, producing a new label. Hash this label to get it into a single label, and repeat for several iterations. The result is a list of aggregated labels that denote graph structure as well as the data stored on the nodes. We found that this measure of similarity made a lot of sense given structure of our Traffic graphs with one central node labeled "device" and the remaining nodes having incoming or outgoing edges to that central node. We chose a standard configuration of 5 iterations.

In addition to these two types of kernels, we computed 5 other widely-used graph kernels found in GraKeL, a package for computing graph kernels in Python[14]. The full list of kernels we computed is: Random Walk, Shortest Path, Weisfeiler-Lehman, Graphlet Sampling, SVM Theta, Neighborhood Hash, and Pyramid Matching. We discovered this package rather late in the process, and due to time constraints did not have time to test every parameter adjustment for all of the kernels, and believe that there may be a more optimal setup. Future work on this project should test out different parameters for the kernels and make use of GraKeL's graph structure that stores node labels, edge labels, and weights. See Table 1 for the results.

For all of these methods we used normalized comparison methods that scaled their output from 1.0 to 0.0. In this way we could confirm when the same graph is compared to itself, a score of 1.0 is produced, and then any structural or attribute/label (in the case of Weisfeiler-Lehman) make the comparison drift closer to 0.0.

Outside the scope of this project, but a highly promising domain for further research is the use of Graph Convolutional Networks[15] or Graph Neural Networks (GNN) for graph classification tasks. GNNs are a relatively new subdomain of deep learning, and more research is needed to determine their feasibility for this task, but we believe this project indicates potential in their use for batched graph classification using node and edge features as described here.

2.5 Transforming Flows into Graph Embeddings

The algorithm we used for converting these flows into graph embeddings filters for the flows corresponding to one device on the network, identifies the unique end-points that this device communicates with based on IP address and ports, and then constructs a directed graph based on the flow of packets from the statistic. Here's an example of the relevant fields from a Netflow Statistic from an HP-Printer:

IP Device: 172.18.153.223
IP Other: 10.1.1.10
Transport Layer Port Device: 62134
Transport Layer Port Other: 53
DNS Hostname: hpb5702b
DNS ClientID: 3173da6fa07d4533f866709086a4c0d510

Given our framework, it was our original intention to store the transport layer ports as features on the nodes, however, given that many services choose random ports, this led to many of the same flows being categorized as different, exponentially increasing the size of our graphs. After realizing this, we decided to disinclude the ports as a part of each node's feature, leaving just the IP address clustered at whichever subnet size is specified as an argument. Additionally, we decided to include a boolean of whether or not to aggregate internal traffic in the tool. For visualization, it is useful to see all of the internal flows, however, when comparing a device to its MUD it is better to combine all flows of the same protocol because the MUD aggregates all local flows to the protocol used. For example, the aforementioned printer's MUD cannot know in advance the IP addresses of devices that will send it documents to print, so it might just say something to the effect of "Allow all internal UDP

traffic". As such, we will want to take all UDP traffic from within the device's own subnet and aggregate it to one node.

3 Experiment

3.1 Part 1: Visualizing Traffic Graphs

The following graph visualizations are from the networkx package with nodes clustered by IP address and protocol from the Brown Guest Netflow dataset. The legends are for identifying the external address or internal address and protocol corresponding to the nodes. To visualize the graph, we use networkx's spring layout format which uses the Fruchterman-Reingold [16] algorithm for force-directed graph embedding. The axis information corresponds to this algorithm, however, it does not have any semantic meaning in our context so you may ignore it. The resulting visualization shows us the distinction between internal and external endpoints (though sometimes in the counter-intuitive way, with external nodes displayed as close to the center). We can tell by the legend which has node label and address if it is an external endpoint, and node label, address, protocol if it is an internal endpoint. Figure 3 shows the graph of a single Amazon Echo's traffic on Brown Guest on a given day clustered by /24. IP addresses beginning with 52 are all part of Amazon's cloud services. In Figure 4, we see that clustering by /16 instead, produces a much smaller graph corresponding to the larger IP spaces that the Amazon echo connects to, many within Amazon's own cloud infrastructure.

Figure 5 shows the graph of an HP Printer on Brown Guest the same day. Notice the size difference of graphs - the HP printer has fewer nodes in the graph because its functionality is relatively simple. This smaller graph is much easier to inspect manually, though. We can see that node 5 is outgoing broadcast traffic (255.255.255.xxx). The flow between node 1 and the device (node 0) is bidirectional. The two internal traffic flows (edges 0,1 and 0,4) are just labeled by their protocols (UDP and TCP respectively). We identify them as internal based on the Netflow data produced from Nick Demarinis's pipeline which identifies IP addresses behind Brown Guest's NAT or not. When a graph is formed without clustering these internal flows (Figure 4), we see all of the different internal TCP and UDP flows to the printer which presumably come from devices sending data to the printer to print.

3.2 Comparing Graphs

To test the viability of this graph embedding for device identification and anomaly detection, we compared a series of ground-truth MUD Graphs from our lab IoT setup with a series of Active Traffic Graphs from the Brown Guest Network. Graph edit distance is discussed in Section 2.4, but we did not include the results here because of their exponential run time and our given time constraints. See Table 1 for summarized results.

The comparisons we made identify a few trends. First, we found that graph edit distance was only possible for the smallest of the graphs - likely because it not feasible to compute in polynomial time. Because of this we omitted the small subset of results we obtained with it. The Weisfeiler-Lehman (WL) algorithm was did produce the highest similarity score between a device's MUD and it's own Active Graph, but not with a much higher score than the other devices. For WL, the best distinguishing power seemed to come for the Chromecast, where it correctly identified it as at least .12 more similar than the other devices.

Some of the measures were not very effective at all with their current settings - the Shortest Path Kernel gave zeros across the board except for the HP-Printer which was 1. Given that shortest path algorithms are pretty similar to graph isomorphism and do not take into account node labels, this makes sense because the MUD and Active Graphs for the Hp Printer were both 7 nodes.

Another promising Kernel was the SVM Theta Kernel[17] which did a pretty good job at identifying devices from their MUD Graphs. The one device this kernel did not perform as well for was the Amazon Echo where it computed that the chromecast's Active Graph was more similar to the MUD Graph of the Echo. That this kernel was otherwise accurate

Random Walk Kernel:

	$MG_{printer}$	MG_{echo}	$MG_{chromecast}$
$AG_{printer}$	**0.9876**	0.9925	0.9904
AG_{echo}	0.9925	**0.9974**	0.9956
$AG_{chromecast}$	0.9920	0.9969	**0.9947**

Shortest Path Kernel:

	$MG_{printer}$	MG_{echo}	$MG_{chromecast}$
$AG_{printer}$	**1.0**	0.0	0.0
AG_{echo}	0.0	**0.0**	0.0
$AG_{chromecast}$	0.0	0.0	**0.0**

Weisfeiler-Lehman:

	$MG_{printer}$	MG_{echo}	$MG_{chromecast}$
$AG_{printer}$	**0.6634**	0.3055	0.4513
AG_{echo}	0.5004	**0.6522**	0.6342
$AG_{chromecast}$	0.6076	0.6280	**0.7511**

Graphlet Sampling Kernel:

	$MG_{printer}$	MG_{echo}	$MG_{chromecast}$
$AG_{printer}$	**1.0**	1.0	1.0
AG_{echo}	1.0	**1.0**	1.0
$AG_{chromecast}$	1.0	1.0	**1.0**

SVM Theta Kernel:

	$MG_{printer}$	MG_{echo}	$MG_{chromecast}$
$AG_{printer}$	**0.9998**	0.4966	0.6603
AG_{echo}	0.0	**0.8001**	0.7477
$AG_{chromecast}$	0.3118	0.9198	**0.8589**

Neighborhood Hash:

	$MG_{printer}$	MG_{echo}	$MG_{chromecast}$
$AG_{printer}$	**0.0909**	0.0	0.0
AG_{echo}	0.0	**0.0**	0.0
$AG_{chromecast}$	0.0	0.0153	**0.1276**

Pyramid Matching:

	$MG_{printer}$	MG_{echo}	$MG_{chromecast}$
$AG_{printer}$	**0.1666**	0.0	0.0
AG_{echo}	0.0	**0.0**	0.0
$AG_{chromecast}$	0.0	0.0314	**0.2791**

Table 1: Graph Kernel comparisons for 3 devices found in both IoT Lab and Brown Guest. MG_{device} indicates a column for the MUD graph for a device and AG_{device} indicates a row for the Active Graph for a device

is an unexpected result given that the SVM Theta Kernel assumes that the graphs are unlabeled. It's accuracy may be due to the difference in size between the graphs of the three devices. Testing on more devices will be necessary to determine the feasibility of this Kernel for this graph classification task.

4 Discussion

4.1 Device complexity

IoT devices have a wide range of uses, some as simple as pulling the weather down from a single website every hour, some nearly as complicated as a computer or smart phone. An important finding of this project was the impact of device complexity on the effectiveness of analysis on its graph embedding. Here we define complexity as the number of different endpoints in its active traffic graph - similarly, the UNSW team uses the length of device MUDs to measure the complexity of a device's network behavior. Contrary to our expectations, we found that more complex devices were actually easier to classify using non-isomorphism based methods. Upon further examination, this is actually a logical result because with more endpoints and features in the graph, there are more possible nodes and edges to match on between devices with similar features, even if the total number or specific IP addresses varied.

More research is needed to determine to the correct level of complexity to produce good results for graph based comparison methods. We suspect that it would fall somewhere in between the simple functionality of a printer, and the almost web browser-like functionality of a Google chromecast. Narrowing in on the right type of device to do this type of analysis on will also aid manufacturers in specifying the class of device they are releasing. Is it closer to simple sensor, or a smartphone. Graph analysis could hold the key to making these determinations - if comparisons are not accurate then it may be too simple or too complex of a device to perform the comparisons on.

4.1.1 Close Look at HP Printer Graphs

Here we examine more closely the Active Graph and MUD Graph generated from data for an HP Printer. This device's MUD is quite short, and as a result, it's graphs are simple and manageable to analyze by hand, Figure X shows the MUD Graph from an HP Printer in UNSW's IoT lab, next to is is the graph formed from an HP Printer on Brown Guest. Both graphs have 6 nodes total, including the device in the center that is connected to all of the other devices. This ismorphic similarity is what causes the shortest path kernel to evaluate to 1.0 - all the possible paths in the two graphs are the same. With non-external traffic clustered by protocol, we see that the HP printer on Brown Guest uses both TCP and UDP internally, presumably devices sending documents to print. The MUD graph however does not contain any TCP flows, only UDP is included in the internal rules. This may be because the printer model is different between the datasets - the Brown Guest Netflow does not tell us the specific device, while the HP Printer whose traffic is used to generate the MUD Graph is known to be DesignJet 70 Printer based on the MAC prefix that the UNSW IoT lab published on their website [7].

The common subnetwork they contact, 15.72.255.xxx resolves to an HP web server, *txe01hpiibpe.ams.hp.net* that the devices share. As part of the safelist, we would expect this flow to be present for the device in practice, and this indicates more than one HP printer use that address space for communication with HP's cloud services.

In terms of differences the Active Graph shows a node corresponding to Brown University's DNS server,(reverse DNS lookup resolves to*bruns2.brown.edu*), while in the MUD Graph the corresponding node is associated with 239.255.255.xxx which resolves to *sns.dns.icann.org*, a root level DNS server. The MUD Graph also contains a node for 192.168.1.0, the typical address of a gateway router in a simple NAT setup. Given that Brown Guest (and other enterprise network setups) are more complicated than a typical NAT, future work on this topic should process MUDs into graphs such that their general or nonapplicable rules are translated into the appropriate form for the given enterprise network. In this case, if the network administration server processing these statistics knows that this printer is connected to gate way with example address *123.456.789.1*, then it should translate the rule for 192.168.1.0 in MUD, into this more locally specific address. In this way, MUD can be parsed into a more relevant form for a network with known configuration.

4.2 Future Work

Much remains to be done in this field of study - this project lays the groundwork for the theoretical approach of using graphs to represent network traffic for Internet of Things devices on enterprise networks. Eventually and pending industry adoption of MUD, this technique could be very useful for determining whether devices have been compromised or not. There is potential for this work to be utilized in the development of an enterprise network security system that keeps track of all the IoT devices on a network and quarantines them if their graph differs too much from MUDs downloaded from the device vendors. A sample workflow for this type of system is diagrammed in Figure 2 Diving more into the realm of deep learning with graphs is also an emerging field and as new developments come forward it may become easier to perform deep-learning graph comparisons on more robust feature vectors encoded on the graph.

Deep Graph Library (DGL)[18] is a new python package that performs deep learning tasks on graph-structured data using a pytorch backend. DGL's documentation provides examples of batched graph similarity testing, however, this does not make use of edge features where we would store connection information. Better integration of the DGL system would suit this project very well with adjusted forward and backward algorithms for training and testing that take into account the way that our data is structured. Along with this comes better vector encodings of the features we identified in this project. For example, for our final implementation we did not use server-side ports as part of the graph embedding, however, UNSW's IoT Lab[7] identifies a bag-of-words model

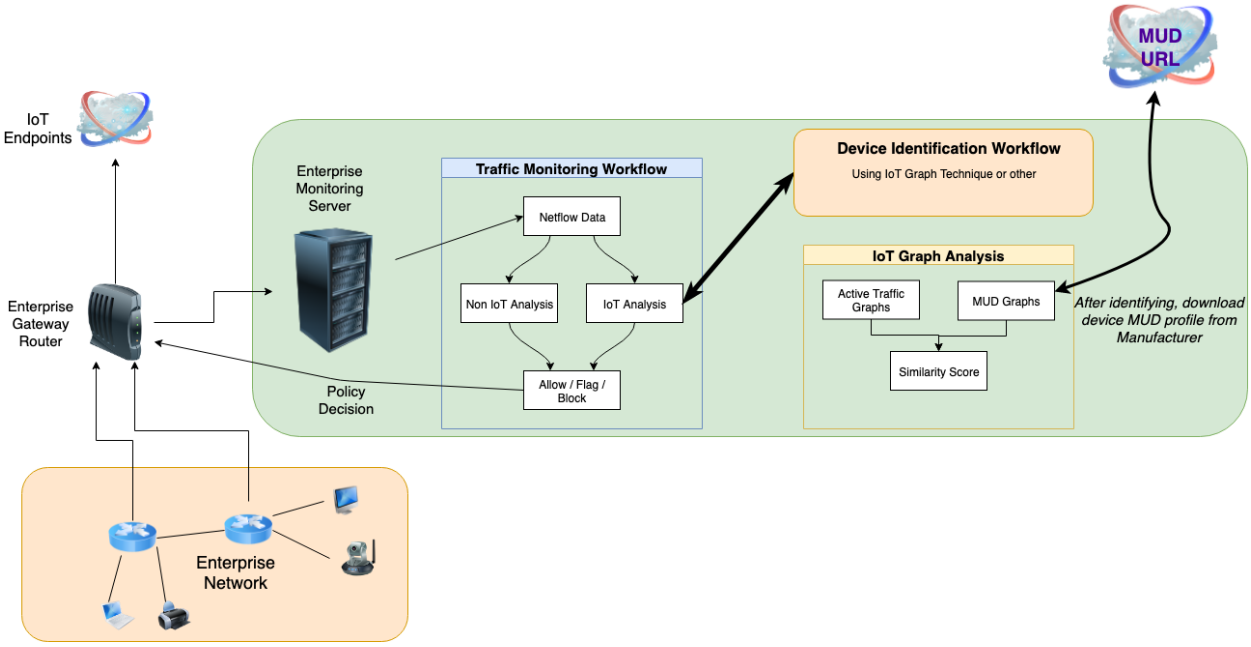

Figure 2: Workflow for Future Deployment of Anomaly Detection System using IoT Graph Analysis

for labeled instances of present port numbers an effective model for encoding this part of the feature space for an Iot Device.

Additionally, for future work on this topic a larger dataset of IoT devices with ranging degrees of functionality complexity would be useful - we were limited to the devices known to be on the Brown Guest network and in our lab. The confirmed overlap between our two data sources was small, and so it did not allow for classification across a wider range of devices. Our original intention was to capture PCAP of the specific devices in Brown University's IoT Lab, but change their configuration to use Brown Guest as their Wifi network. In this way then, it would be possible to isolate a specific device's flows in the enterprise Netflow data. Unfortunately, due to time and hardware-related technical constraints, we were unable to connect our devices to Brown Guest while also capturing their traffic on a mirroring server, so we ended up using PCAP from similar devices from UNSW's IoT lab. While we know some of the devices to be very similar (Amazon Echo, Chromecast), the specific HP printer model was likely different between the datasets, as well as potentially different firmware versions for all the devices. A future project might solve this problem, or obtain Netflow from a privately managed network and convert it to the enterprise network style flows that we used here.

Future collaborators are encouraged to contribute to our code [6], to update the feature generation portion of the pipeline, and to try out different graph comparison methods on the MUD and Active Graphs. Given the success of graph embeddings for non-IoT traffic in network security, we see a lot of potential for this work to be expanded upon to more accurately classify IoT devices based on their graphs.

5 Conclusion

This project finds promising results for IoT Device network analysis by way of using graph representations of their traffic patterns. The organized structure of a graph is very effective for storing the infor-

mation relevant to an IoT Device's traffic flows, and we found that graph comparison methods demonstrate potential for comparing graphs for the purpose of identifying devices or detecting malicious traffic. While our specific implementation does not provide enough comparison power to definitively determine what device a traffic pattern belongs to, or if that device is behaving maliciously, it certainly points to the need for more research in this domain, especially as the field of machine learning on graph-like data structures continues to develop as well. We stress the importance of choosing feature sets to use to encode traffic information on nodes and edges of the graph structure. We suspect our modest results are due to the incompatibility of how we encoded this information on the nodes to be used in the Graph Kernel Computations. With the labels as part of a more coherent alphabet, and the correct balance of generalized vs. specific features encoded in the labels could lead to more definitive comparison results. Deep learning also holds great potential to classify graph structured data more effectively than the traditional methods we tried here, and we feel this is the logical next direction for this research to go.

6 Acknowledgements

Special thanks to Theophilus Benson, Nicholas DeMarinis, Rodrigo Fonseca and the rest of the Brown University Department of Computer Science for advising, supporting, and encouraging this Master's research project.

References

[1] T. Benson and B. Chandrasekaran, "Sounding the bell for improving internet (of things) security," in *Proceedings of the 2017 Workshop on Internet of Things Security and Privacy*, ser. IoTS&P '17. New York, NY, USA: ACM, 2017, pp. 77–82. [Online]. Available: http://doi.acm.org/10.1145/3139937.3139946

[2] "Abstract." [Online]. Available: https://tools.ietf.org/id/draft-ietf-opsawg-mud-22.html

[3] F. Mansmann, F. Fischer, D. Keim, and S. North, "Visual support for analyzing network traffic and intrusion detection events using treemap and graph representations," 01 2009.

[4] S. Boddy and J. Shattuck, "The hunt for iot: The rise of thingbots," Aug 2017. [Online]. Available: https://www.f5.com/labs/articles/threat-intelligence/the-hunt-for-iot-the-rise-of-thingbots

[5] A. Hamza, D. Ranathunga, H. Habibi Gharakheili, M. Roughan, and V. Sivaraman, "Clear as mud: Generating, validating and applying iot behavioral profiles," 08 2018, pp. 8–14.

[6] "Iot graph analysis repo," access Permission may need to be requested. [Online]. Available: https://systems-git.cs.brown.edu/iotsec/iotsec

[7] A. Sivanathan, H. Habibi Gharakheili, F. Loi, A. Radford, C. Wijenayake, A. Vishwanath, and V. Sivaraman, "Classifying iot devices in smart environments using network traffic characteristics," *IEEE Transactions on Mobile Computing*, pp. 1–1, 2018.

[8] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and P. S. Yu, "A comprehensive survey on graph neural networks," *CoRR*, vol. abs/1901.00596, 2019. [Online]. Available: http://arxiv.org/abs/1901.00596

[9] X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, and D. Song, "Neural network-based graph embedding for cross-platform binary code similarity detection," *CoRR*, vol. abs/1708.06525, 2017. [Online]. Available: http://arxiv.org/abs/1708.06525

[10] M. R. Garey and D. S. Johnson, *Computers and Intractability; A Guide to the Theory of NP-Completeness*. New York, NY, USA: W. H. Freeman & Co., 1990.

[11] S. V. N. Vishwanathan, K. M. Borgwardt, I. R. Kondor, and N. N. Schraudolph, "Graph kernels," *CoRR*, vol. abs/0807.0093, 2008. [Online]. Available: http://arxiv.org/abs/0807.0093

[12] K. M. Borgwardt and H. P. Kriegel, "Shortest-path kernels on graphs," in *Fifth IEEE International Conference on Data Mining (ICDM'05)*, Nov 2005, pp. 8 pp.–.

[13] N. Shervashidze, P. Schweitzer, E. J. van Leeuwen, K. Mehlhorn, and K. M. Borgwardt, "Weisfeiler-lehman graph kernels," *J. Mach. Learn. Res.*, vol. 12, pp. 2539–2561, Nov. 2011. [Online]. Available: http://dl.acm.org/citation.cfm?id=1953048.2078187

[14] G. Siglidis, G. Nikolentzos, S. Limnios, C. Giatsidis, K. Skianis, and M. Vazirgiannis, "Grakel: A graph kernel library in python," *arXiv preprint arXiv:1806.02193*, 2018.

[15] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," *CoRR*, vol. abs/1609.02907, 2016. [Online]. Available: http://arxiv.org/abs/1609.02907

[16] T. M. J. Fruchterman and E. M. Reingold, "Graph drawing by force-directed placement," *Software: Practice and Experience*, vol. 21, no. 11, pp. 1129–1164, 1991. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.4380211102

[17] F. Johansson, V. Jethava, D. Dubhashi, and C. Bhattacharyya, "Global graph kernels using geometric embeddings," in *Proceedings of the 31st International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, E. P. Xing and T. Jebara, Eds., vol. 32, no. 2. Bejing, China: PMLR, 22–24 Jun 2014, pp. 694–702. [Online]. Available: http://proceedings.mlr.press/v32/johansson14.html

[18] Dmlc, "dmlc/dgl," May 2019. [Online]. Available: https://github.com/dmlc/dgl

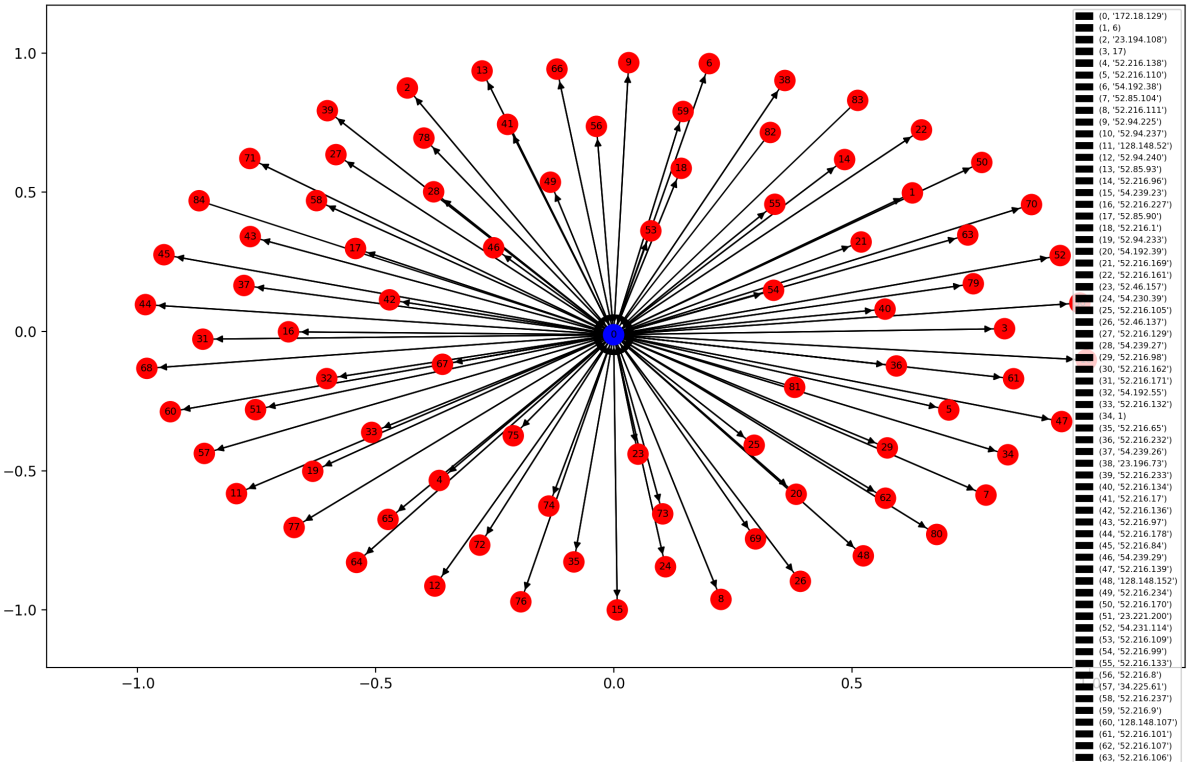

Figure 3: Active Traffic Graph from an Amazon Echo on Brown Guest Wifi network. Legend corresponds to the IP address of the nodes or the protocol of internal IP flows

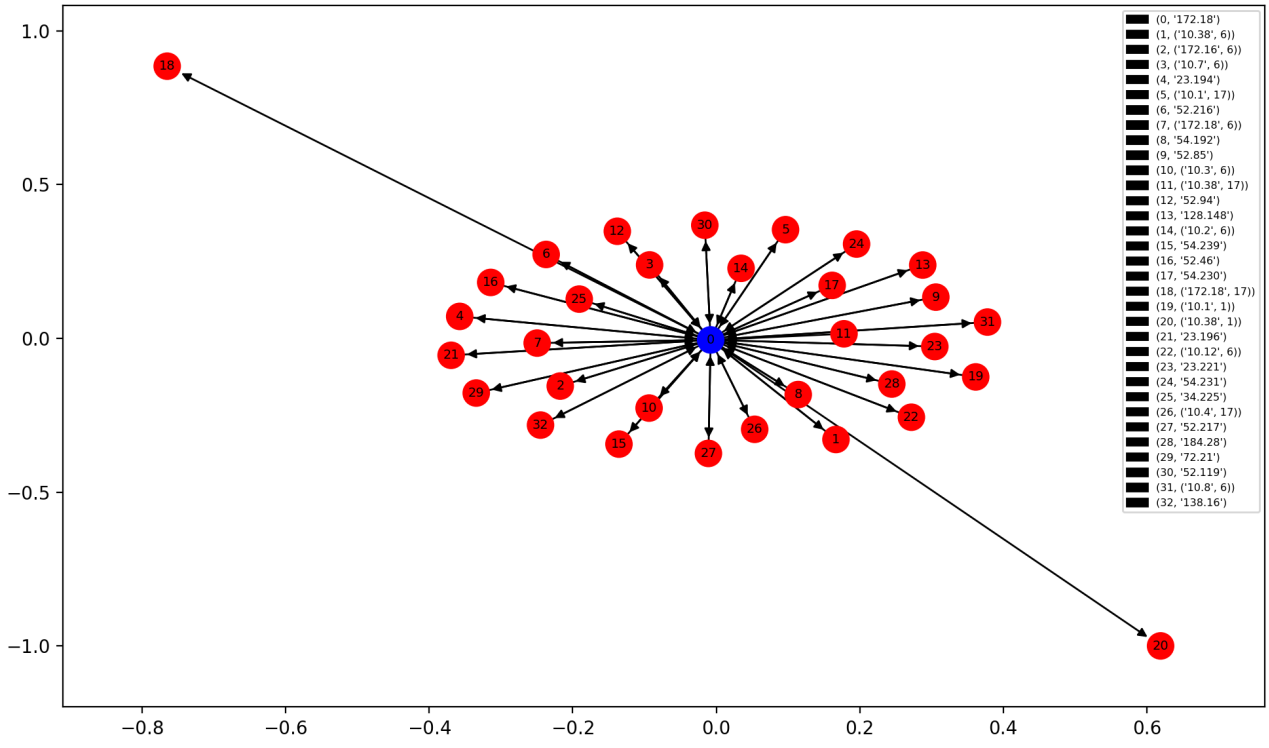

Figure 4: Active Traffic Graph from an Amazon Echo on Brown Guest Wifi network. Legend corresponds to the IP address of the endpoints. This figure includes internal traffic not clustered by protocol.

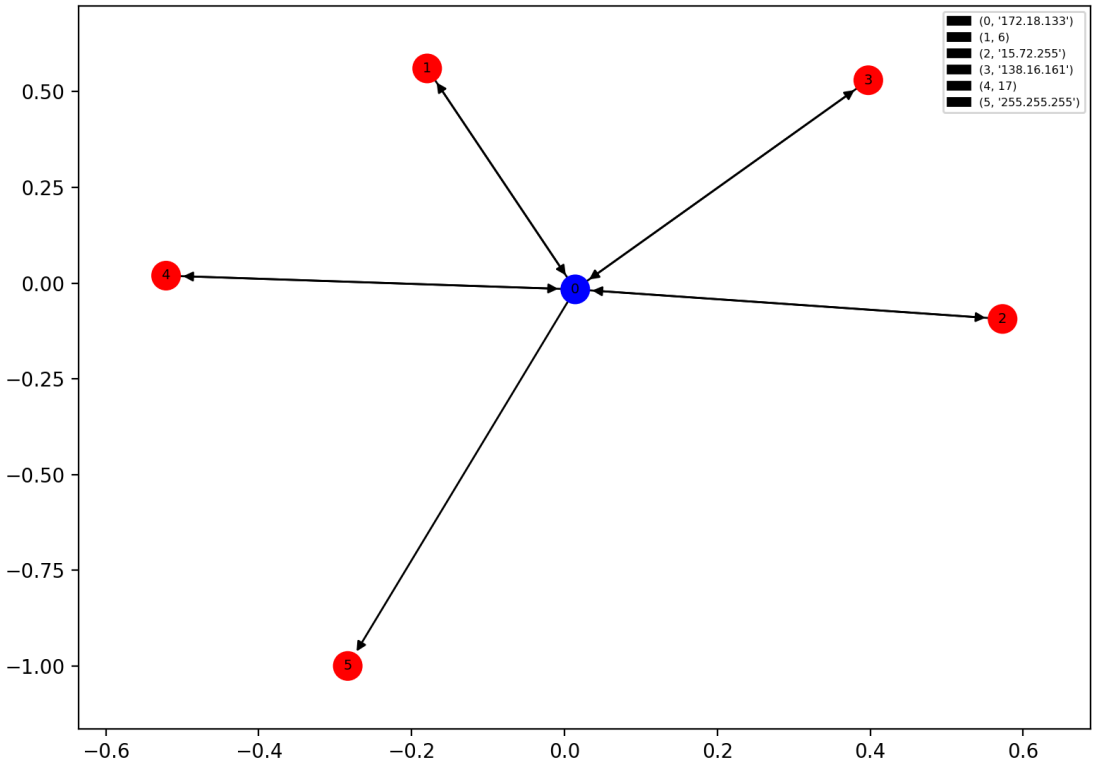

Figure 5: Active Traffic Graph from an HP Printer on Brown Guest Wifi network. Legend corresponds to the IP address of the nodes or the protocol of internal IP flows

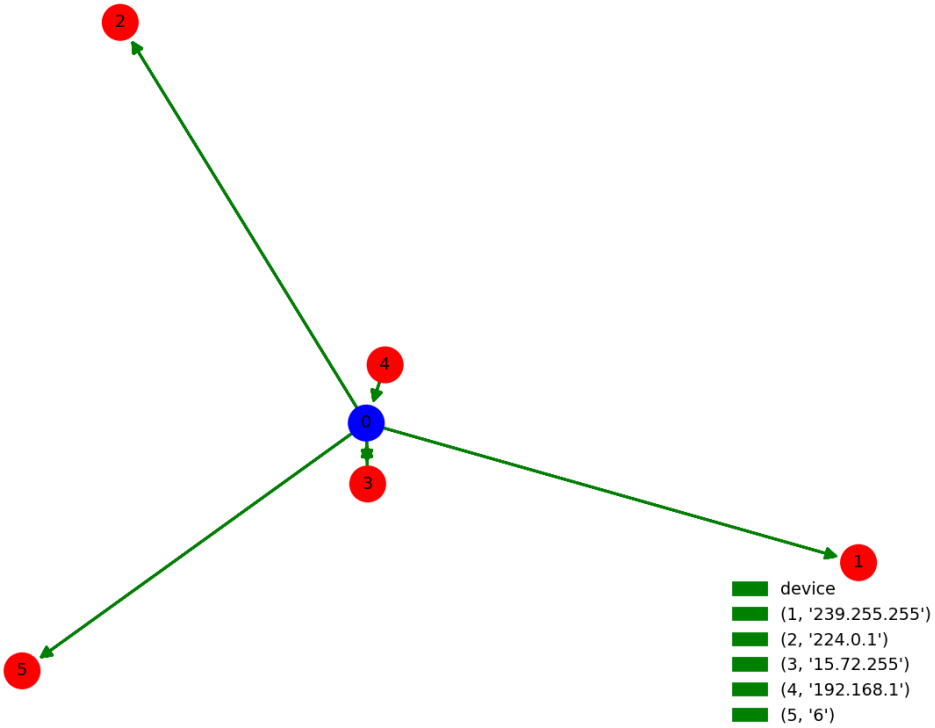

Figure 6: MUD Graph from an HP Printer in UNSW's IoT lab. Legend corresponds to the IP address of the nodes or the protocol of internal IP flows

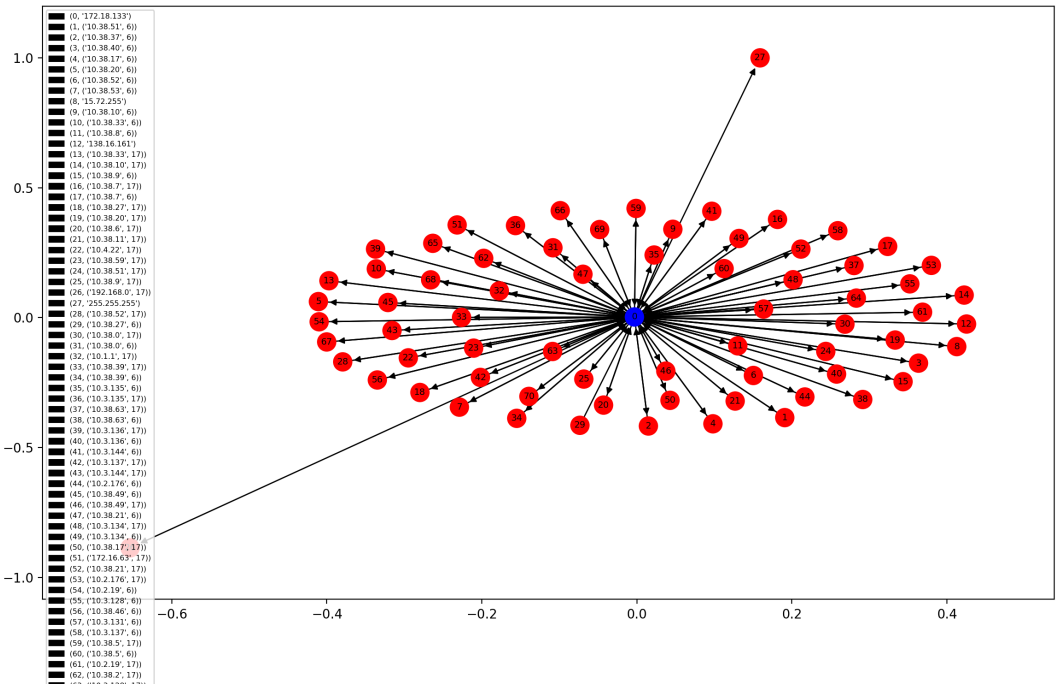

Figure 7: Active Traffic Graph from an HP Printer on Brown Guest Wifi network. Legend corresponds to the IP address of the endpoints without clustering by protocol. Here we see all of the devices using the printer to print.