New Capabilities for Self-Driving Networks

Nolan Rudolph
University of Oregon
ngr@uoregon.edu

ABSTRACT

Granted the annual trends in increasing internet usage, the University of Oregon Networking Research Group preemptively researches the concept of Self-Driving Networks (S-DNs) to create a self-remediating, high-performance network. In efforts of accomplishing this project, the lack of S-DN compatible software compels new research to be conducted on new capabilities for a self-driving network. In this project, we accomplish a light-weight visualization framework for flow level data accompanied by a scalable flow to packet generator usable by S-DNs.

1 INTRODUCTION

As the usage of internet able devices continues to increase each successive year [1][2][3], so does the demand for secure, scalable, and high-performance networks. Granted these trends of internet usage, the University of Oregon Networking Research Group aims to create a networking device that is entirely self-reliant and able to operate at unprecedented speeds, without the scrutinization of a network administrator. This Self-Driving Network (S-DN) would allow for astronomical advancements in the networking community and act as a first step in the creation of a globally robust and impenetrable internet.

A device capable of operating at such high speeds requires state of the art software to enable such performance. In light of these missing necessities, my research project is centered around the creation of three new capabilities for S-DNs: (1) A visualization framework for real-time depictions of network traffic, (2) the ability to actively generate packets from monitored flows, and (3) creating numerous APIs for future network administrators to change network policies.

Since S-DNs emphasize the importance of low overhead and the general optimization of its services, the main forwarding structure used for the

transportation of user data within a S-DN is analogous to Cisco's NetFlow [4] -- A CSV style representation of network traffic flows which characterizes communications into its most salient attributes (e.g. layer protocols, MAC addresses). This is because flow level data is notorious for its miniscule byte count, simplistic flow accumulation methods, and its compatibility with the vast majority of data recording and forwarding network devices [5][6].

Therefore, in order for my capabilities to be compatible with S-DNs, I had to revolve my project around the usage of flow level data. Furthermore, as optimization was key for the success of my project, I decided to use C programming as my coding language of choice to provide myself more control over the architecture and system interactions of my programs.

2 LITERATURE REVIEW

2.1 General Computer Networking. Before I began coding, I realized I didn't know much about network packets and had no prior experience in network programming. Luckily, Ross's *Computer Networking* textbook [7] was all I needed to get my project underway. In this text, Ross speaks of the OSI model of networking, a seven layer depiction of how a packet can be constructed and transmitted along different mediums and then deconstructed at alternate endpoints in order to efficiently transfer data. Fortunately, Ross talks in depth about the application, transport, network, and link layers which perfectly coincided with the intentions of my project.

1. Link Layer: Once a packet is put on a medium (e.g. fiber optic cables), the link layer is comprised of switches which utilize data frames from packets to forward data to the correct location.
2. Network Layer: Similar to the Link Layer, the Network Layer utilizes a particular internet protocol specified by the packet to guide the data to the appropriate location.
3. Transport Layer: Here, the method of transportation specified by the packet is used to forward the data in a particular way, be it maintaining a continuous connection between two endpoints or one-way communications.
4. Application Layer: Lastly, this is where the packet finishes deconstructing and presents the payload, or important data of the packet, to the requesting user.

These layers are referred to as layer 1, layer 2, layer 3, and layer 7, respectively. Therefore, I knew I had to translate flow level data into packets which were comprised of these four different layers.

2.2 Program Optimization. Continuing this idea of overall optimization, I looked to generic programming textbooks for guidance. One in particular caught my attention, O'Hallaron's *Computer Systems* textbook, which brought me great insight into a programmer's perspective of code optimization. In this text I learned numerous development techniques that help decrease the number of cycles a computer has to make to run code:

1. Eliminating Loop Inefficiencies: Direct Memory Accessing (DMA) costs computers a considerable amount of processing power and time. Therefore, if there is any way to remove memory references (e.g. using a temporary local variable), then that should be implemented
2. Reducing Procedure Calls: Not only do function calls utilize DMA, but they also run through a set of instructions that can cost many cycles. It is wise to remove gratuitous function calls from repetitive procedures
3. Loop Unrolling: This can be utilized to accomplish multiple iterations in a single iteration inside a loop, removing the need to continuously reference and write to the same memory addresses
4. Program Parallelism: Enhancing parallelism in ones program allows a new process to begin before the last one finishes. This way, no locks will be encountered in critical sections of code.

I would keep all of these strategies in mind throughout the entirety of writing my own program.

Unfortunately for my research, creating visualization frameworks, packet generators, and APIs for flow level data is such an esoteric subject that it was difficult to find general help for my envisioned software. Therefore, most information that guided me to my end product wasn't found in a book but rather small excerpts from websites such as StackOverflow where I applied other people's issues and resolutions to my own questions at hand.

3 METHODS

3.1 Dataset Acquisition and Discoveries.

I was fortunate enough to acquire a dataset of 280 million recorded flows at the University of Oregon to use for my own testing purposes. Among these reported flows they contained the following categories delimited by commas:

1. Start Timestamp in Epoch Format
2. End Timestamp in Epoch Format
3. Source IP Address
4. End IP Address
5. Source Port Number
6. Destination Port Number (or a float type.code if ICMP/IGMP/IPv6 ICMP)
7. IP Protocol Number
8. Type of Service (ToS)
9. Transmission Control Protocol Flags (defaulted to 0 if IP protocol is not TCP)
10. Number of Packets
11. Number of Bytes
12. Router Ingress Port
13. Router Egress Port
14. Source ASN
15. Destination ASN

Upon dissection of this data, I found that the top four protocols associated with the IP protocol field were ICMP, IGMP, TCP, and UDP. Since including all 255 protocols would be a tedious and not so prosperous task for my research, I decided to implement only these four protocols for my future flow to packet synthesizer.

With this surplus of data at my disposal, I was ready to get started on my visualization framework.

3.2 Visualization Framework

The first step in achieving a visualization framework for my new flow data was to find a scalable and efficient visualization tool for my data.

I began by discovering a program named *Gephi* [8] which gives a topology type reputation of any data you grant it. Unfortunately, the end result was disastrous and gave no insight to the data. Furthermore, a single compile of data took over an hour to complete, which is not practical for my project.

Next, I turned my attention to *Excel* [9] to see if there was any applicability to visualizing real-time flow level data. With small datasets, Excel was excellent at gathering all sorts of statistics about my data and even had the ability to produce graphs at a rapid pace, ranging from line charts to stack graphs. Unfortunately, Excel comes with a table cap of 1.04M entries which isn't

feasible with the number of flows I would realistically have to visualize per second. In the instance of a Ducks Football game, a million entries would be considered unusually small, rendering this visualization method incompatible.

Granted that Excel is generally used for business analytics, I decided to try out Excel's counterpart, *Tableau* [10]. Tableau grants the visualization aspect that Excel tends to lack, and allows for customizable visualizations of data. Fortunately, Tableau has no maximum entry cap, however upon attempting to upload 50 million entries, Tableau quickly became unresponsive. Upon testing a 1 million entry dataset, the compile took around 30 seconds which was not quick enough for my requirements.

Finally, I came across *Grafana* [11], a visualization software that binds itself to a port and allows you to display data from a SQL server. Immediately I was enthralled by these services, being as my dataset was delimited by commas and SQL databases allow CSV imports. After binding Grafana to one of my ports, I decided to create a SQL server initialized with the following table:

```
CREATE TABLE UONETFLOW(
      startTime double,
      endTime double,
      srcIP varchar(16),
      dstIP varchar(16),
      srcPort varchar(5),
      destPort varchar(5),
      IPProt varchar(3),
      TOSVal varchar(2),
      TCPFlags varchar(3),
      packets int(8),
      bytes int(16),
      routerInPort varchar(5),
      routerOutPort varchar(5),
      srcASN varchar(5),
      dstASN varchar(5)
);
```

*Note: These variables will be referenced throughout the remainder of this section. I then imported the University of Oregon network flow dataset into this table, and the outcome was a desirable final product.

3.3 Flow to Packet Generator

We now approach the phase where my prior readings and programming methods take effect. I began by constructing simple packets with layer 1 being an Ethernet frame, layer 2 an IP header, layer 3 the UDP header, and a gibberish payload for layer 7. The netinet repository [12] from the FreeBSD C library provided me with all the headers I required, equipped with compact structures used as layer headers. Sockets [13], provided by the GNU C Library, allowed me to reliably transfer packets onto the transmission queue of my NIC for packet sending. After tinkering with my program for

some time, I found I could reliably transfer data over the localhost interface using my prototype [14].

Unfortunately, the localhost interface is a very simplistic feedback loop interface which is an unreliable means of testing a program's efficiency and viability. Therefore, I decided to transfer my tests to CloudLabs [15], a cloud provisioning platform that grants you access to a multitude of nodes for network testing. After pulling my GitHub repository [14] onto my client node and issuing a TCPdump on a linked interface on my server node, I found that if I specified source and destination MAC addresses alongside a network interface, my protocol could reliably transfer packets from client to server!

With my prototype finished, it wasn't too difficult to implement the rest of the protocols since all I had to do was replace the UDP header with a desired header, and configure it according to the flow entry in the dataset. Once I had implemented ICMP, IGMP, and TCP, I was ready to move onto time related issues.

The question became: How could I read flows from a dataset and send packets that replicated the flow over a set duration of time? I began by implementing this concept of having my program's runtime correlate with the time from the dataset. I did this by creating a method which reads the first entry from the dataset, grabs the start time of the packet, and subtracts it from every subsequent flow in the dataset. This way, all flow times have been neutralized to program time, which makes it much simpler to decide when to read a new flow; My program should only read a new flow if the start time is before or equal to my program's current time.

Now that my program knows when to read flows, it must know when to send from the flow as well. I achieved this goal by taking the nettime the flow was recorded for and dividing that value by the net amount of packets the flow had, `(endTime - startTime) / packets`. This was stored as variable `d_time`, where I discovered I needed to create a specific structure that would encompass details about the flow for low overhead sending. Therefore, I decided that when each flow is read, it would then be stored as a `grand_packet` structure with the following attributes:

1. `char *buffer` -- Used for storing the contents of the actual packet itself

2. `unsigned int packets_left` -- Tracks how many packets are left in the entry
3. `float d_time` -- Holds the calculated delta time between packet transmission, calculated via the method shown above
4. `double cur_time` -- Tracks the current time state of the packet for scheduling
5. `length` -- Holds the total length of the packet for socket transmission
6. `struct grand_packet *last` -- Used for network scheduling
7. `struct grand_packet *next` -- Used for network scheduling

Now, we can use the program's time in contrast with the flow's `cur_time` to determine if another packet should be sent. If the flow should be sent at some program run-time, then a packet is sent using the flow's `buffer` and we add the flow's `d_time` to its `cur_time`, 1 is subtracted from `packets_left`, and then program continues to monitor the flow. Once the flow's `packets_left` reaches 0, the memory addresses that held the flow's structure are freed.

Lastly, we must now consider that the dataset will have many flows whose start time will fall within the program's current time -- I found a max of half a million flows that fell victim from this particular dataset! Therefore, I needed some sort of scheduler that would check each applicable flow and see if it was time for that flow to send, do nothing, or free its own memory. I resolved this issue by applying a Round-Robin (RR) algorithm to network scheduling, where each flow would be designated a small amount of time, where the RR would then continue onto the next flow in the circle. Therefore, when each new flow is implemented to the RR circle, I set the first flow's `*last` to point to the new flow, and the last flow's `*next` to point to the new flow as well. This way, I have a linked list of flows that the RR can easily traverse and send from when required.

3.4 APIs for Future Network Administrators

Unfortunately, the visualization framework and flow to packet generator took up the entirety of the time I had to conduct research in. Please read the FUTURE WORK section for more information on what I intend to accomplish on this front in the near future.

4 RESULTS

4.1 Visualization Framework

After saturating the SQL database and linking Grafana's services, the end result was

a desirable outcome.

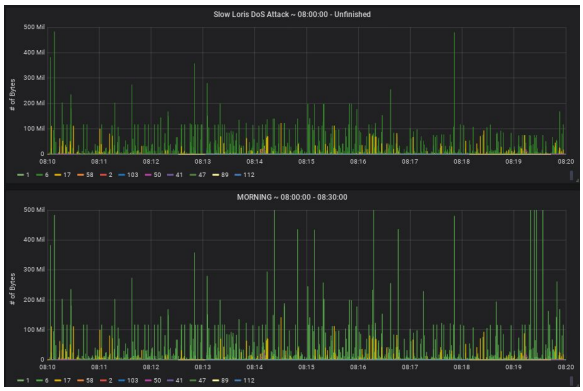

Figure 1: Comparing two ten minute periods from my UO dataset

Here, one could easily visualize the ongoing communications in one's network. After testing different portions of the vast dataset with Grafana's services, I was able to find spikes and trends in the graphs.

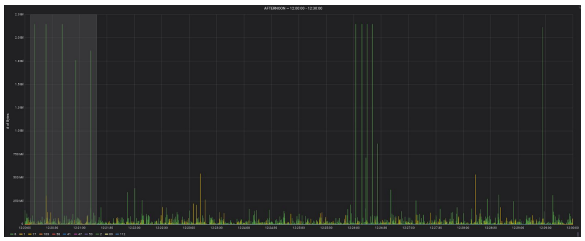

Figure 2: Isolating spiked region from visualized portion of UO dataset

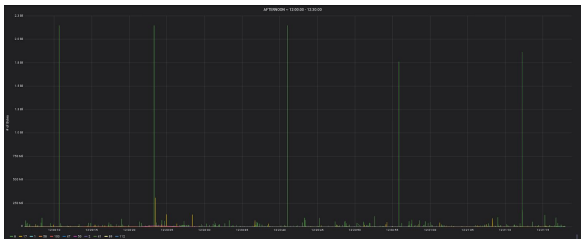

Figure 3: Resulting graph from enlargement

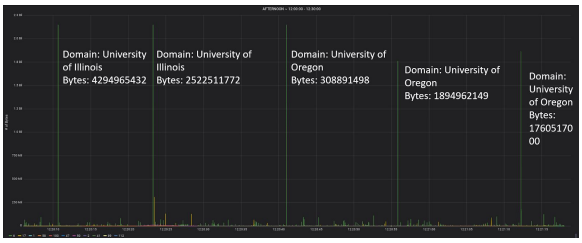

Figure 4: Domain and byte labels applied to anomalies in Figure 3

I later discovered that Grafana's services could be automated which further supports the idea of being usable with S-DNs, which will be explained further in FUTURE WORKS.

4.2 Flow to Packet Generator

My flow to packet generator software never fell behind when generating packets for the dataset I supplied it. Therefore, I decided to test the limits of my program to get a range of benchmarks for potential usages.

I began by creating test flows for my software using a lightweight flow generating script [14]. This script was utilized by automated testing, a bash shell script I created which creates flows of byte rates up to link-rate (generally 10 Gb/s for most NICs) [14]. The performance of my flow to packet generator, called *UONetflowC*, are shown below.

Figure 5: UONetflowC PPS, run on Intel Xeon D-1548 2.0 GHz core, MLNX X-3 10 Gb/s NIC

Here, we can see that my program tends to decline in accuracy around 900 kpps. Granted the number of packet options my program accommodates for, how it runs entirely in kernel space, and that I use a single core to transmit all packets, these results are rather impressive! In fact, my results compete fairly well with some of the leading kernel space packet synthesizers, such as *TCPReplay* [16].

Figure 5: TCPReplay run on Intel Xeon D-1548 2.0 GHz core, MLNX X-3 10 Gb/s NIC

We witness that TCPReplay's max PPS is around 950 kpps, and on a single core, 825 kpps. Therefore, my program takes the win on a single core, however without multithreading capabilities, it falls short in a multi-core comparison.

I ran another test to see what sort of bit rates my flow to packet generator could achieve. All packets must abide by an MTU of 1500, therefore a realistic max packet size that wouldn't be dropped by the kernel is approximately 1300 bits. With this method, I was able to achieve line-rate BPS on my node's 10 Gb/s NIC as shown below.

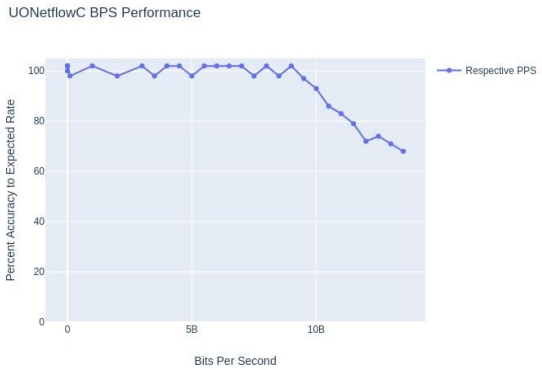

Figure 6: UONetflowC BPS, run on Intel Xeon D-1548 2.0 GHz core, MLNX X-3 10 Gb/s NIC

5 CONCLUSION

For my project, I successfully created a visualization framework to be used by S-DNs in order to visualize any state of network traffic. The refresh rate for the

visualizer to update its graph is sub 5 seconds, making it an excellent tool for real-time monitoring of S-DNs. This framework can also assist network administrators in identifying anomalies and adding new policies to their network to avoid further attacks.

Furthermore, I created a flow to packet generator capable of reading from flow datasets and generating realistic packets accordingly. The generator uses a system of adjusting recorded flow time frames to that of the program's time, and sending in a fashion that perfectly replicates each flow. The flow to packet generator was able to reach a single core, kernel space packets per second of 900 kpps, which competes extremely well against other state of the art kernel space packet generators.

6 FUTURE WORK

I want to begin by this section by stating my project is nowhere near complete, but this research period served as an excellent starting point for the rest of the goals I would like to accomplish.

6.1 Future of the Visualization Framework

Although I was able to create a solid method of viewing network flow data, there are no real-time applications at this current moment. There is no way to link my visualization framework to a S-DN without a multitude of scripts, designed in the Go scripting language as Grafana's API is in Go. Therefore, the next step I'll take in implementing a visualization framework for S-DNs is allowing the linking between a S-DN's incoming network flows and a real-time visual monitor of such flows.

6.2 Future of the Flow to Packet Generator

Creating this flow to packet generator in a low level language such as C has allowed me to conceptualize what needs to be achieved in order to reliably transmit packets over a network interface. Unfortunately, my main fault in the creation of my packet generator was using kernel space API (e.g. sockets offered by GNU C library) since there is a performance hindering overhead involved with calling kernel functions.

After conducting research on a multitude of state of the art packet generators such as Netmap [17], MoonGen [18], and Pktgen-DPDK [19], I've found that there are frameworks these packet generators use to bypass the kernel space and allow direct access to NIC drivers through their own respective APIs. The four frameworks I

decided to research were PF_RING ZC, Netmap, DPDK, and Snabb. Upon conducting more elaborate research, I've found that DPDK and Snabb have had the best results in fast packet processing [20][21]. Therefore, I intend to begin by implementing the Snabb into my own project to boost my performance benchmarks, and if dissatisfied with the results, I would then turn my attention to the DPDK framework.

6.3 APIs for Network Administrators

Since the flow to packet generator took a large chunk of my research time, I wasn't able to create new APIs for future network administrators. However, I can pinpoint exactly how I intend on doing so. I will be creating an API that directly influences this framework and the output of flows. Flows can be isolated through my API and defining particular attributes of a flow for simple viewing, thresholds for byte counts can be set to view anomalies, and one can use these options to decide whether or not a flow should bypass a network filter. I intend on implementing this API after I use the Snabb framework with my flow to packet generator, and have linked that to my visualization framework.

7 ACKNOWLEDGEMENTS

Before concluding this paper, I would be remiss if not to acknowledge Chris Misa, an important member of the University of Oregon Networking Research Group, and his huge contribution to this project. We met once a week for the duration of this project where he assisted me in resolving a bunch of issues that I incurred within the previous week.

I would also like to thank my mentor Ramakrishnan Durairajan for making this whole opportunity a possibility for me. Words cannot explain how thankful I am to be working on such interesting projects as these.

Last and certainly not least, I would like to give my greatest gratitude to the University of Oregon Undergraduate Research Opportunity Program for their VPRI Fellowship, sponsoring all the work shown in this project. I would not have been able to accomplish what I have without their help, and for this I am beyond grateful.

8 REFERENCES

[1] "ICT Statistics of 2001-2018," *Statistics*. [Online]. Available: https://www.itu.int/en/ITU-D/Statistics/Pages/stat/default.aspx. [Accessed: 20-Sep-2019].

[2] "Internet Growth Statistics 1995 to 2019 - the Global Village Online," *Internet World Stats*. [Online]. Available: https://www.internetworldstats.com/emarketing.htm. [Accessed: 20-Sep-2019].

[3] E. Ortiz-Ospina, "The rise of social media," *Our World in Data*. [Online]. Available: https://ourworldindata.org/rise-of-social-media. [Accessed: 21-Sep-2019].

[4] "Introduction to Cisco IOS NetFlow - A Technical Overview," *Cisco*, 29-May-2012. [Online]. Available: https://www.cisco.com/c/en/us/products/collateral/ios-nx-os-software/ios-netflow/prod_white_paper0900aecd80406232.html. [Accessed: 16-Sep-2019].

[5] B. Davenport, "A Comprehensive List of Cisco NetFlow Capable Devices," *Plixer*, 27-Sep-2017. [Online]. Available: https://www.plixer.com/blog/comprehensive-list-cisco-netflow-capable-devices/. [Accessed: 16-Sep-2019].

[6] "sFlow Able Network Equipment," *sFlow Products: Network Equipment @ sFlow.org*. [Online]. Available: https://sflow.org/products/network.php. [Accessed: 16-Sep-2019].

[7] J. F. Kurose and K. W. Ross, *Computer networking: a top-down approach*. Hoboken, NJ: Pearson, 2017.

[8] "The Open Graph Viz Platform," *Graph Exploration and Manipulation*. [Online]. Available: https://gephi.org/. [Accessed: 26-Sep-2019].

[9] "Microsoft Excel," *Spreadsheet Software*. [Online]. Available: https://products.office.com/en-us/excel. [Accessed: 26-Sep-2019].

[10] "Tableau: Business Intelligence and Analytics Software," *Tableau Software*. [Online]. Available: https://www.tableau.com/. [Accessed: 26-Sep-2019].

[11] "Grafana: The open observability platform," *Grafana Labs*. [Online]. Available: https://grafana.com/. [Accessed: 26-Sep-2019].

[12] Netinet. https://github.com/afabbro/netinet

[13] GNU C Library. Socket. https://github.com/torvalds/linux/blob/master/include/linux/socket.h

[14] Nolan Rudolph. UONetflowC. https://github.com/NolanRudolph/UONetflowC

[15] University of Utah, "Build Your Own Cloud on Our Hardware," *CloudLab*. [Online]. Available: https://cloudlab.us/. [Accessed: 26-Sep-2019].

[16] "Pcap Editing and Replaying Utilities," *Tcpreplay*. [Online]. Available: https://tcpreplay.appneta.com/. [Accessed: 27-Aug-2019].

[17] Luigi Rizzo. Netmap. https://github.com/luigirizzo/netmap.

[18] Paul Emmerich. MoonGen. https://github.com/emmericp/MoonGen.

[19] Keith Wiles. Pktgen-DPDK. http://github.com/Pktgen/Pktgen-DPDK

[20] P. Emmerich, S. Gallenmüller, D. Raumer, F. Wohlfart, and G. Carle, "MoonGen: A Scriptable High-Speed Packet Generator," 28-Oct-2015. [Online]. Available: https://www.net.in.tum.de/fileadmin/bibtex/publications/papers/MoonGen_IMC2015.pdf. [Accessed: 16-Sep-2019].

[21] A. Wingo, "Production High-Performance Networking with Snabb and LuaJIT," *LinkedIn*, 24-Jan-2017. [Online]. Available: https://www.slideshare.net/igalia/production-highperformance-networking-with-snabb-and-luajit-linuxconfau-2017. [Accessed: 13-Sep-2019].