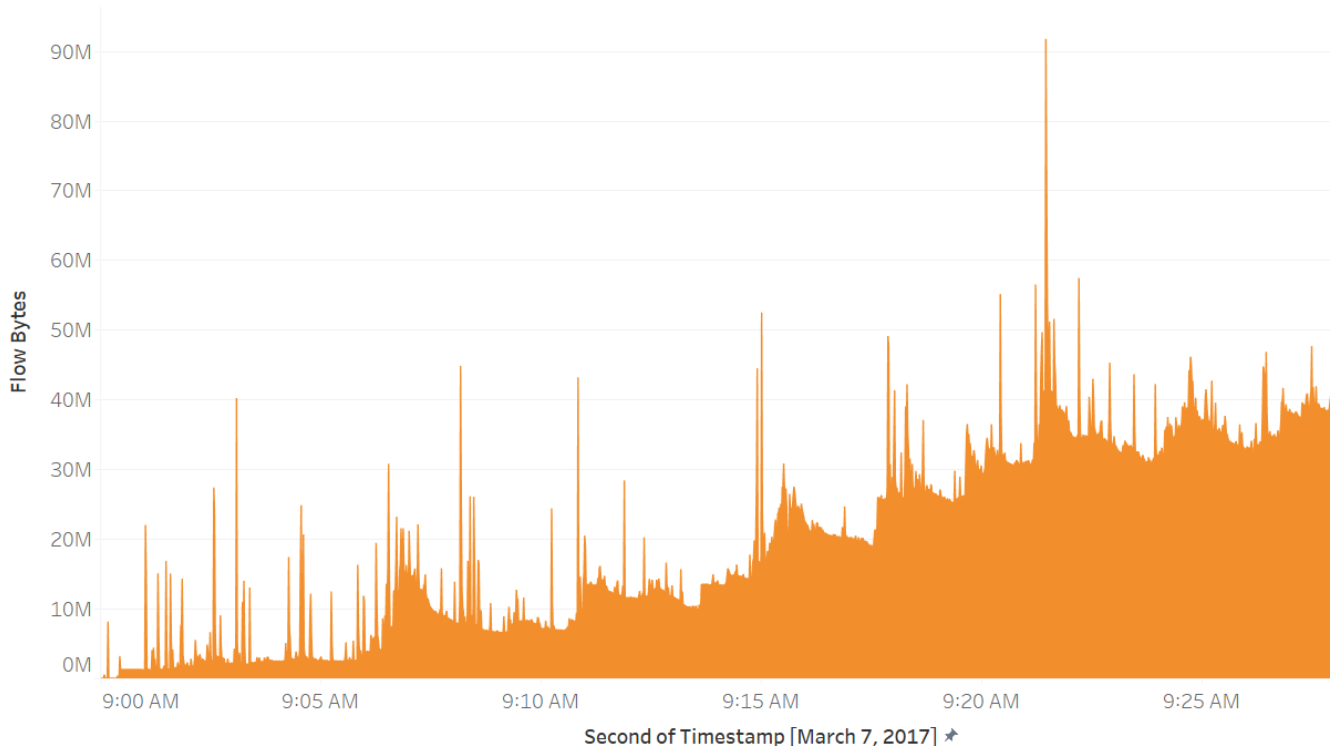


# PCAP stream processing

## Overview

This project is a proof of concept of real-time parsing and storage of enterprise network traffic. Processing network traffic flow is an unstructured data problem, in which 10's or 100's of megabytes could be easily passing a sensor per second (Figure 1). Exploring such technical infrastructure is important for real-time machine learning applications for detecting cyber exploits. However, PCAP is only one data source among several streaming sources that can enrich cyber operations analyses.



**Figure 1:** Packet bytes per second from an analysis of the CICDataset's Labelled Flows (Sharafaldin, Lashkari, and Ghorbani, 2018). The analysis calculated the average bytes per second of the full volume of the flow via the timestamp, flow duration, and flow bytes per second features in the flow-level traffic report. See *Monday-WorkingHours.pcap\_ISCX.csv* in [data/](#) sourced from *GeneratedLabelledFlows.zip* at [205.174.165.80/CICDataset/CIC-IDS-2017/Dataset/](https://205.174.165.80/CICDataset/CIC-IDS-2017/Dataset/).

The datasets used in this project are PCAP files of capture data of generated network activity. [The Intrusion Detection Evaluation Dataset](#) (CICIDS2017) is hosted by the University of New Brunswick: Canadian Institute for Cybersecurity. The dataset consists of full packet payloads in pcap format, along with labeled network flows and extracted datasets for machine learning (sets of packet exchanges between hosts). However, for this particular project, the interest is in the original packet activity with the passage of time. Each file was downloaded to simulated switch server with the `wget` command.

```
[{ "file": "Monday-WorkingHours.pcap",
  "url": "http://205.174.165.80/CICDataset/CIC-IDS-2017/Dataset/PCAPs/Monday-WorkingHours.pcap",
  "GB": "10",
  "exploit": "benign"},
{ "file": "Wednesday-WorkingHours.pcap",
  "url": "http://205.174.165.80/CICDataset/CIC-IDS-2017/Dataset/PCAPs/Wednesday-WorkingHours.pcap",
  "GB": "12",
  "exploit": "DDoS attack"}]
```

## Architecture overview

Without access to an enterprise network switch, this project simulates the traffic by replaying the CICDataset on a cloud server, which listens to the traffic and sends unstructured packet lines to a Spark cluster for processing. The master node then stores the lines transformed as key-value pairs (Figure 2).

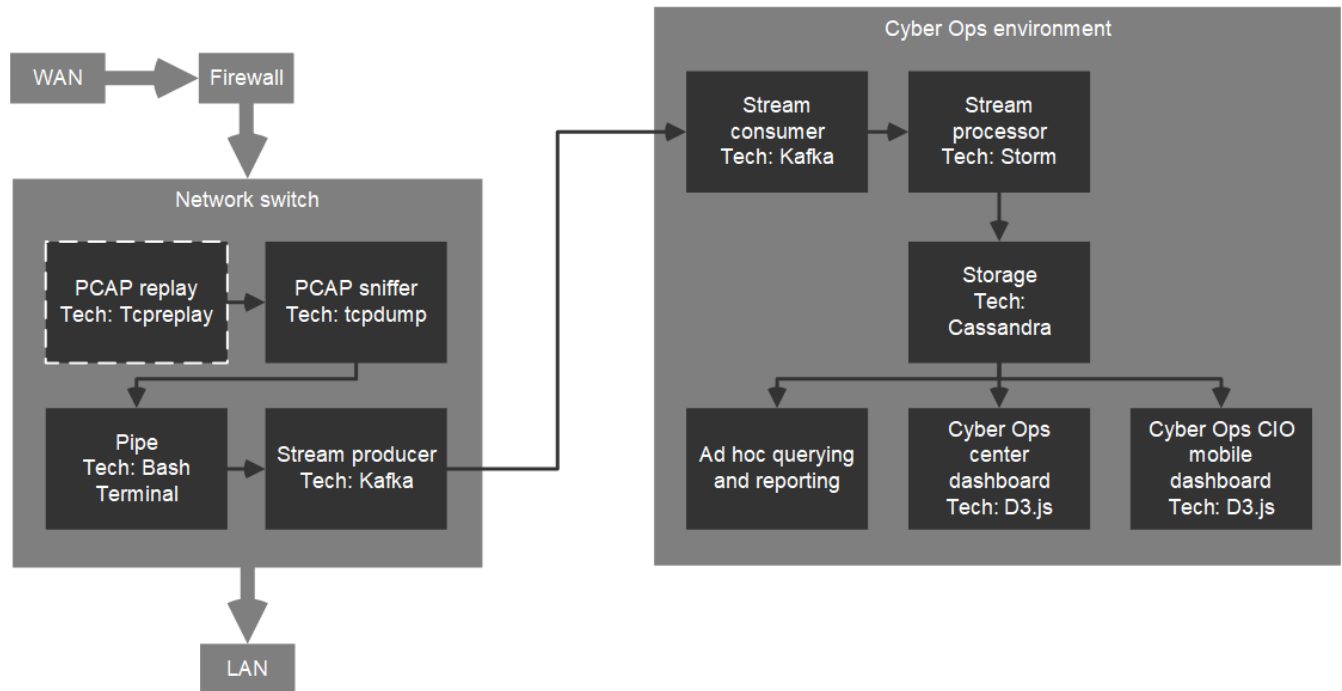


Figure 2

## Network switch

The simulated network switch, named instance-1, replays the PCAP files with the command line tool tcpreplay. "The basic operation of tcpreplay is to resend all packets from the input file(s) at the speed at which they were recorded, or a specified data rate, up to as fast as the hardware is capable" (tcpreplay man page, Debian). Another common networking tool, tcpdump captures the replayed packets. (Note that the tool also captures actual instance-1 SSH session traffic, which could be filtered out.) The standard output is piped to yet another networking tool, netcat. Netcat serves the data on a specified port for consumption of the Spark cluster. It may be noted that netcat is a quick solution and that a queuing or messaging service ought to be implemented. A service like Kafka provides scalability and high-availability message queues on clusters, with a level of security. The following configuration includes the installation of Kafka on instance-1; however it is not implemented in the current project version.

## Configuration

- Infrastructure: [Google Cloud Platform Compute Engine](#)
- Name: instance-1
- Operating system: Debian
- vCPU: 1\*
- Memory: 3.75 GB
- Disk 30GB

\*tcpreplay's CPU usage (97%) probably calls for increasing the number of cores (Figure 3).

```
top - 04:15:09 up 53 min, 5 users, load average: 0.58, 0.63, 0.42
Tasks: 93 total, 2 running, 91 sleeping, 0 stopped, 0 zombie
%Cpu(s): 99.7 us, 0.3 sy, 0.0 ni, 0.0 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st
KiB Mem : 3792960 total, 119480 free, 693308 used, 2980172 buff/cache
KiB Swap: 0 total, 0 free, 0 used, 2832896 avail Mem

  PID USER      PR  NI  VIRT  RES  SHR  S %CPU  %MEM    TIME+  COMMAND
 7411 root        20   0 13092 2120 1932 R 96.7  0.1   0:06.39 tcpreplay
3876 kafka      20   0 4166624 351880 18976 S 1.7  9.3   0:14.24 java
3517 oberljn    20   0 42828 3552 2984 R 0.3  0.1   0:01.63 top
6327 root        20   0 0 0 0 S 0.3  0.0   0:00.35 kworker/u2:2
6685 kafka      20   0 2506492 126896 19040 S 0.3  3.3   0:03.78 java
7049 kafka      20   0 2508548 116312 18904 S 0.3  3.1   0:03.78 java
 1 root        20   0 56872 6652 5268 S 0.0  0.2   0:01.00 systemd
 2 root        20   0 0 0 0 S 0.0  0.0   0:00.00 kthreadd
 3 root        20   0 0 0 0 S 0.0  0.0   0:00.06 ksoftirqd/0
 5 root        0 -20 0 0 0 S 0.0  0.0   0:00.00 kworker/0:0H
 6 root        20   0 0 0 0 S 0.0  0.0   0:00.96 kworker/u2:0
 7 root        20   0 0 0 0 S 0.0  0.0   0:00.10 rcu_sched
 8 root        20   0 0 0 0 S 0.0  0.0   0:00.00 rcu_bh
 9 root        rt   0 0 0 0 S 0.0  0.0   0:00.00 migration/0
10 root        0 -20 0 0 0 S 0.0  0.0   0:00.00 lru-add-drain
11 root        rt   0 0 0 0 S 0.0  0.0   0:00.00 watchdog/0
12 root        20   0 0 0 0 S 0.0  0.0   0:00.00 cpuph/0
13 root        20   0 0 0 0 S 0.0  0.0   0:00.00 kdevtmpfs
14 root        0 -20 0 0 0 S 0.0  0.0   0:00.00 netns
15 root        20   0 0 0 0 S 0.0  0.0   0:00.00 khungtaskd
16 root        20   0 0 0 0 S 0.0  0.0   0:00.00 oom_reaper
17 root        0 -20 0 0 0 S 0.0  0.0   0:00.00 writeback
18 root        20   0 0 0 0 S 0.0  0.0   0:00.00 kcompactd0
20 root        25   5 0 0 0 S 0.0  0.0   0:00.00 ksm
21 root        39  19 0 0 0 S 0.0  0.0   0:00.00 khugepaged
22 root        0 -20 0 0 0 S 0.0  0.0   0:00.00 crypto
23 root        0 -20 0 0 0 S 0.0  0.0   0:00.00 kintegrityd
24 root        0 -20 0 0 0 S 0.0  0.0   0:00.00 bioset
25 root        0 -20 0 0 0 S 0.0  0.0   0:00.00 kblockd
26 root        0 -20 0 0 0 S 0.0  0.0   0:00.00 devfreq_wq
27 root        0 -20 0 0 0 S 0.0  0.0   0:00.00 watchdogd
30 root        20   0 0 0 0 S 0.0  0.0   0:01.24 kswapd0
31 root        0 -20 0 0 0 S 0.0  0.0   0:00.00 vmstat
43 root        0 -20 0 0 0 S 0.0  0.0   0:00.00 kthrotld
45 root        0 -20 0 0 0 S 0.0  0.0   0:00.00 ipv6_addrconf
90 root        20   0 0 0 0 S 0.0  0.0   0:00.00 scsi_eh_0
93 root        0 -20 0 0 0 S 0.0  0.0   0:00.00 scsi_tmf_0
106 root        0 -20 0 0 0 S 0.0  0.0   0:00.00 bioset

Warning in send_packets.c:send_packets() line 178:
Unable to send packet: Error with PF_PACKET send() [251]: Message too long (errno = 90)
Warning in send_packets.c:send_packets() line 178:
Unable to send packet: Error with PF_PACKET send() [255]: Message too long (errno = 90)
Warning in send_packets.c:send_packets() line 178:
Unable to send packet: Error with PF_PACKET send() [256]: Message too long (errno = 90)
Warning in send_packets.c:send_packets() line 178:
Unable to send packet: Error with PF_PACKET send() [258]: Message too long (errno = 90)
Warning in send_packets.c:send_packets() line 178:
Unable to send packet: Error with PF_PACKET send() [259]: Message too long (errno = 90)
Warning in send_packets.c:send_packets() line 178:
Unable to send packet: Error with PF_PACKET send() [261]: Message too long (errno = 90)
Warning in send_packets.c:send_packets() line 178:
Unable to send packet: Error with PF_PACKET send() [262]: Message too long (errno = 90)
Warning in send_packets.c:send_packets() line 178:
Unable to send packet: Error with PF_PACKET send() [264]: Message too long (errno = 90)
Warning in send_packets.c:send_packets() line 178:
Unable to send packet: Error with PF_PACKET send() [265]: Message too long (errno = 90)
Warning in send_packets.c:send_packets() line 178:
Unable to send packet: Error with PF_PACKET send() [267]: Message too long (errno = 90)
Warning in send_packets.c:send_packets() line 178:
Unable to send packet: Error with PF_PACKET send() [268]: Message too long (errno = 90)
Warning in send_packets.c:send_packets() line 178:
Unable to send packet: Error with PF_PACKET send() [270]: Message too long (errno = 90)
Warning in send_packets.c:send_packets() line 178:
Unable to send packet: Error with PF_PACKET send() [271]: Message too long (errno = 90)
Warning in send_packets.c:send_packets() line 178:
Unable to send packet: Error with PF_PACKET send() [273]: Message too long (errno = 90)
^C
Actual: 287 packets (155581 bytes) sent in -1588824605.79 seconds.          Rated:
-0.0 bps, -0.00 Mbps, -0.00 pps
oberljn@instance-1:~/pcap_streaming$ ip link list |grep eth0
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1460 qdisc pfifo_fast state UP mode DEFA
ULT group default qlen 1000
oberljn@instance-1:~/pcap_streaming$ sudo ip link set eth0 mtu 1500
oberljn@instance-1:~/pcap_streaming$ sudo tcpreplay -i eth0 -K --loop 1 smallFlows.pcap
sending out eth0
processing file: smallFlows.pcap
^C
Actual: 8047 packets (4937706 bytes) sent in -1588824676.23 seconds.          Rated:
-0.0 bps, -0.00 Mbps, -0.00 pps
oberljn@instance-1:~/pcap_streaming$ sudo tcpreplay -i eth0 -K --loop 1 smallFlows.pcap
sending out eth0
processing file: smallFlows.pcap
```

Figure 3 The left session running the "top" command shows the CPU usage of tcpreplay. It is undetermined what results from tcpreplay maxing out the CPU, such as dropped or delayed packets.

## Resize the disk

Storing the two CICDataset PCAP files requires at least 25GB available. The disk must be resized and filesystem extended (Resizing the file system, Google Cloud).

Instance's disk page -> Edit -> input new size -> Save

Check disks with

```
sudo df -h
sudo lsblk
```

Make sure growpart from cloud-guest-utils is installed:

```
sudo apt-get install cloud-guest-utils
```

resize with device ID and partition number.

```
sudo growpart /dev/sda 1
```

Extend the file system in order to use the additional space.

```
sudo resize2fs /dev/sda1
```

## Install tcpdump, tcpreplay, and Java JDK

Java is required for Kafka.

```
sudo apt-get update
sudo apt-get upgrade
```

```
sudo apt-get install tcpreplay
sudo apt-get install tcpdump
sudo apt install default-jdk
```

## Install and configure Kafka

Make Kafka user and give sudo privileges.

```
sudo useradd kafka -m
sudo passwd kafka
sudo adduser kafka sudo
```

Log into user and download and extract the Kafka binaries. Check that the binary version exists at [downloads.apache.org/kafka](https://downloads.apache.org/kafka).

```
su -l kafka
mkdir ~/Downloads
curl "https://downloads.apache.org/kafka/2.5.0/kafka_2.13-2.5.0.tgz" -o ~/Downloads/kafka.tgz
mkdir ~/kafka
cd ~/kafka
tar -xvzf ~/Downloads/kafka.tgz --strip 1
```

Configure the Kafka server by editing its properties.

```
vim ~/kafka/config/server.properties
```

Add to the end of the file:

```
delete.topic.enable = true
```

Create the systemd unit files for zookeeper and Kafka. Upload [zookeeper.service](#) and [kafka.service](#) to the instance. Copy to system directory.

```
sudo cp ../oberljn/zookeeper.service /etc/systemd/system/zookeeper.service
sudo cp ../oberljn/kafka.service /etc/systemd/system/kafka.service
```

This daemon reload may be necessary.

```
sudo systemctl daemon-reload
```

Start servers.

```
sudo systemctl start zookeeper
sudo systemctl start kafka
```

Check that Kafka is running.

```
sudo journalctl -u kafka
```

Enable Kafka on boot.

```
sudo systemctl enable kafka
```

Make a heartbeat topic that will be part of the cyber operation's monitoring of its infrastructure. Such a topic might include timestamped CPU and memory usage stats via a tool like sysstat.

```
~/kafka/bin/kafka-topics.sh --create --zookeeper localhost:2181 --replication-factor 1 --partitions 1 --topic instan
```

This Python script `kafka_heartbeat.py` is a wrapper around the Kafka publish command that sends a stats message every N seconds. (Figure 4)

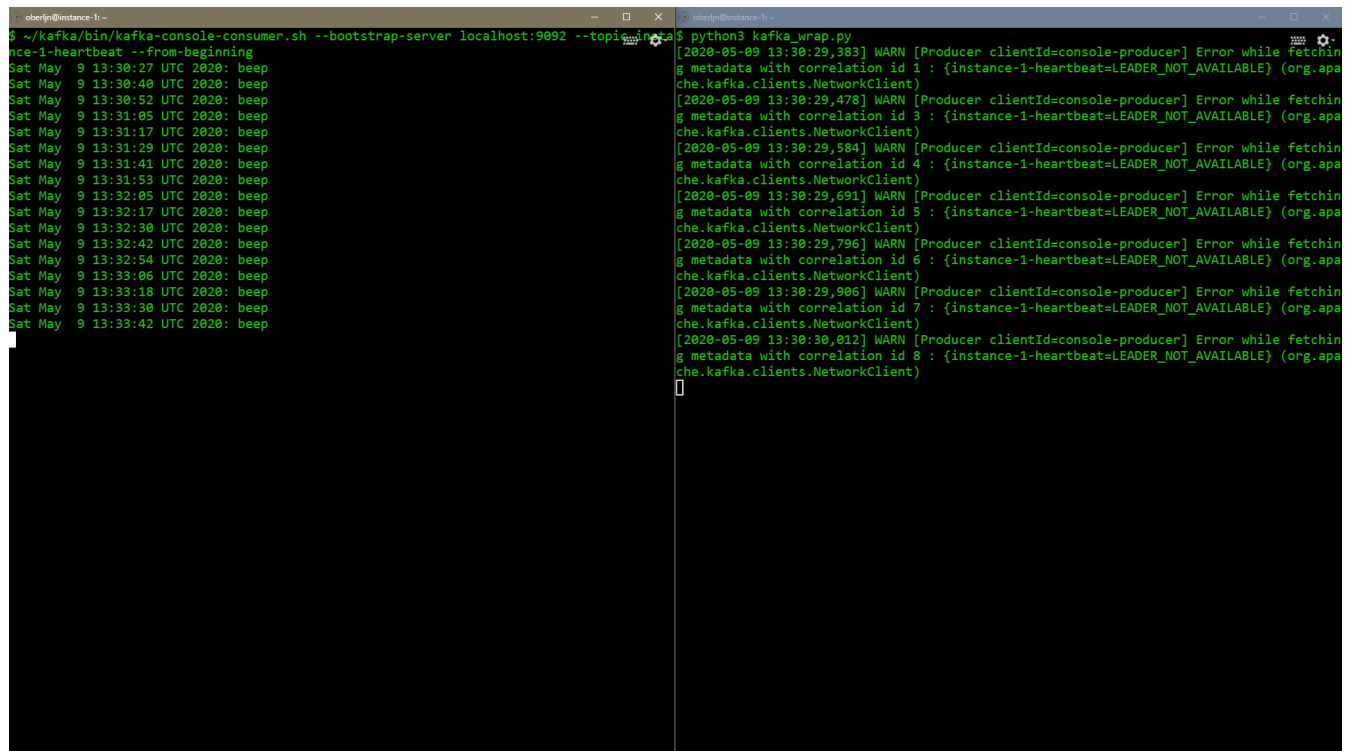


Figure 4 shows two terminal windows. The left window displays the output of a Kafka console consumer, showing a series of 'beep' messages with timestamps. The right window shows the output of a Python script, displaying warning messages from the Kafka producer client.

```
~/kafka/bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic instance-1-heartbeat --from-beginning
Sat May 9 13:30:27 UTC 2020: beep
Sat May 9 13:30:40 UTC 2020: beep
Sat May 9 13:30:52 UTC 2020: beep
Sat May 9 13:31:05 UTC 2020: beep
Sat May 9 13:31:17 UTC 2020: beep
Sat May 9 13:31:29 UTC 2020: beep
Sat May 9 13:31:41 UTC 2020: beep
Sat May 9 13:31:53 UTC 2020: beep
Sat May 9 13:32:05 UTC 2020: beep
Sat May 9 13:32:17 UTC 2020: beep
Sat May 9 13:32:30 UTC 2020: beep
Sat May 9 13:32:42 UTC 2020: beep
Sat May 9 13:32:54 UTC 2020: beep
Sat May 9 13:33:06 UTC 2020: beep
Sat May 9 13:33:18 UTC 2020: beep
Sat May 9 13:33:30 UTC 2020: beep
Sat May 9 13:33:42 UTC 2020: beep

$ python3 kafka_wrap.py
[2020-05-09 13:30:29,383] WARN [Producer clientId=console-producer] Error while fetching metadata with correlation id 1 : {instance-1-heartbeat=LEADER_NOT_AVAILABLE} (org.apache.kafka.clients.NetworkClient)
[2020-05-09 13:30:29,478] WARN [Producer clientId=console-producer] Error while fetching metadata with correlation id 3 : {instance-1-heartbeat=LEADER_NOT_AVAILABLE} (org.apache.kafka.clients.NetworkClient)
[2020-05-09 13:30:29,584] WARN [Producer clientId=console-producer] Error while fetching metadata with correlation id 4 : {instance-1-heartbeat=LEADER_NOT_AVAILABLE} (org.apache.kafka.clients.NetworkClient)
[2020-05-09 13:30:29,691] WARN [Producer clientId=console-producer] Error while fetching metadata with correlation id 5 : {instance-1-heartbeat=LEADER_NOT_AVAILABLE} (org.apache.kafka.clients.NetworkClient)
[2020-05-09 13:30:29,796] WARN [Producer clientId=console-producer] Error while fetching metadata with correlation id 6 : {instance-1-heartbeat=LEADER_NOT_AVAILABLE} (org.apache.kafka.clients.NetworkClient)
[2020-05-09 13:30:29,906] WARN [Producer clientId=console-producer] Error while fetching metadata with correlation id 7 : {instance-1-heartbeat=LEADER_NOT_AVAILABLE} (org.apache.kafka.clients.NetworkClient)
[2020-05-09 13:30:30,012] WARN [Producer clientId=console-producer] Error while fetching metadata with correlation id 8 : {instance-1-heartbeat=LEADER_NOT_AVAILABLE} (org.apache.kafka.clients.NetworkClient)
```

Figure 4

## Running the traffic simulation, capture, and serving

It was found that, to use Kafka, tcpdump is piped to netcat, which pipes to the Kafka producer script for the topic. For example, after making a topic "instance-1-pcap", in an instance-1 session, sniff network traffic with tcpdump and pipe to port 4444 with netcat.

```
sudo tcpdump -i eth0 -nn -v | netcat localhost 4444
```

Here, `-i` to point tcpdump to the ethernet interface `eth0`, `-nn` to disable IP and port name resolution, and `-v` to increase the verbosity of the output.

In another session, listen on port 4444 with netcat and pipe lines to Kafka.

```
netcat -l -p 4444 | ~/kafka/bin/kafka-console-producer.sh --broker-list localhost:9092 --topic instance-1-pcap > /dev
```

However, due to errors when trying to consume the Kafka topic from the Spark cluster, and a lack of time for troubleshooting, this project opts to use only netcat to serve the data, for illustrative purpose.

Along with the tcpdump and netcat commands above, tcpreplay is ran on a third session:

```
sudo tcpreplay -i eth0 -K --loop 1 smallFlows.pcap
```

The `--netmap` switch will cause `tcpreplay` to write to network buffers directly, bypassing the network driver if it is detected. "This will allow you to achieve full line rates on commodity network adapters, similar to rates achieved by commercial" (`tcpreplay` man page, Debian).

The three commands are wrapped in `server_threads.py`, which also includes the configuration of the instance's maximum transmission unit. The MTU is the size of the largest protocol data unit that is allowed to be transmitted. One can check the data flow on another session with the following. Note, this command must run after running `server_threads.py`, specifically after `netcat -lk -p 4444`.

```
netcat localhost 4444
```

```
oberlin@instance-1:~$ tcpdump -i eth0 -s 262144 -C 1000 -w - 'port 5555'
tcpdump: listening on eth0, link-type EN10MB (Ethernet), capture size 262144 bytes
14:04:02.277359 IP (tos 0x0, ttl 128, id 30523, offset 0, flags [DF], proto TCP (6), length 40)
    192.168.3.131.58265 > 208.82.236.129.80: Flags [F.], cksum 0xec38 (correct), seq 607844800, win 0, len 0
14:04:02.326523 IP (tos 0x0, ttl 242, id 36462, offset 0, flags [DF], proto TCP (6), length 40)
    208.82.236.129.80 > 192.168.3.131.58265: Flags [.], cksum 0x18e8 (correct), seq 665088, win 0, len 0
14:04:02.961927 IP (tos 0x10, ttl 64, id 14914, offset 0, flags [DF], proto TCP (6), length 2888)
    10.150.0.6.22 > 74.125.42.37.44370: Flags [P.], cksum 0x8a78 (incorrect -> 0xa8c5), seq 149057:151893, ack 704, win 296, options [nop,nop,TS val 547256 ecr 2208830613], length 2836
14:04:02.961938 IP (tos 0x10, ttl 64, id 14916, offset 0, flags [DF], proto TCP (6), length 1376)
    10.150.0.6.22 > 74.125.42.37.44370: Flags [P.], cksum 0x8490 (incorrect -> 0x1c6f), seq 151893:153217, ack 704, win 296, options [nop,nop,TS val 547256 ecr 2208830613], length 1324
14:04:02.961970 IP (tos 0x10, ttl 64, id 14917, offset 0, flags [DF], proto TCP (6), length 2888)
    10.150.0.6.22 > 74.125.42.37.44370: Flags [P.], cksum 0x8a78 (incorrect -> 0x34d3), seq 153217:156053, ack 704, win 296, options [nop,nop,TS val 547256 ecr 2208830613], length 2836
14:04:02.961974 IP (tos 0x10, ttl 64, id 14919, offset 0, flags [DF], proto TCP (6), length 1376)
    10.150.0.6.22 > 74.125.42.37.44370: Flags [P.], cksum 0x8490 (incorrect -> 0x41f1), seq 156053:157377, ack 704, win 296, options [nop,nop,TS val 547256 ecr 2208830613], length 1324
14:04:02.962003 IP (tos 0x10, ttl 64, id 14920, offset 0, flags [DF], proto TCP (6), length 2888)
    10.150.0.6.22 > 74.125.42.37.44370: Flags [P.], cksum 0x8a78 (incorrect -> 0x4edb), seq 157377:160213, ack 704, win 296, options [nop,nop,TS val 547256 ecr 2208830613], length 2836
14:04:02.962006 IP (tos 0x10, ttl 64, id 14922, offset 0, flags [DF], proto TCP (6), length 1376)
    10.150.0.6.22 > 74.125.42.37.44370: Flags [P.], cksum 0x8490 (incorrect -> 0x992d), seq 160213:161537, ack 704, win 296, options [nop,nop,TS val 547256 ecr 2208830613], length 1324
14:04:02.962018 IP (tos 0x10, ttl 64, id 14923, offset 0, flags [DF], proto TCP (6), length 276)
    10.150.0.6.22 > 74.125.42.37.44370: Flags [P.], cksum 0x8044 (incorrect -> 0x43f4), seq 161537:161761, ack 704, win 296, options [nop,nop,TS val 547256 ecr 2208830613], length 224
14:04:02.976209 IP (tos 0x10, ttl 64, id 14924, offset 0, flags [DF], proto TCP (6), length 2888)
    10.150.0.6.22 > 74.125.42.37.44370: Flags [P.], cksum 0x8a78 (incorrect -> 0x34d3), seq 161761:164607, ack 704, win 296, options [nop,nop,TS val 547256 ecr 2208830613], length 2836
```

**Figure 5** Session 1 (left) runs `serve_threads.py`, which consists of `tcpreplay`, `tcpdump`, and `netcat`. Session 2 listens with `netcat` and shows the streaming packet capture. This same stream will be consumed by the Spark cluster.

[^ top](#)

## Spark cluster

The Spark cluster processes the PCAP stream of lines consumed from the network switch, instance-1. Its primary transformation is to extract elements from the unstructured stream of lines. The cluster could also be used to perform Natural Language Processing on the packet, in order to classify traffic as an exploit.

## Configuration

Repeat Kafka install steps as in instance-1 configuration.

## Parsing PCAP

using regular expressions to store key traffic data, such as timestamp, source IP, destination IP, protocol, packet size, etc.

See [http\\_request\\_dump.txt](#) in [appendix/](#).

# END

## Consume Kafka topics from instance 1

First, attempt to publish and consume PCAP in Kafka from one instance.

#here Make a PCAP topic.

```
~/kafka/bin/kafka-topics.sh --create --zookeeper localhost:2181 --replication-factor 1 --partitions 1 --topic instan
```

Is netcat needed? Why not tcpdump piped to Kafka producer?

Connection 1: Listen on port with netcat and pipe to Kafka producer.

```
netcat -l -p 4444 | ~/kafka/bin/kafka-console-producer.sh --broker-list localhost:9092 --topic instance-1-pcap > /de
```

Connection 2: Sniff network traffic and pipe to port vi netcat.

```
sudo tcpdump -i eth0 -nn -v | netcat localhost 4444
```

Connection 3: Replay the network traffic

```
sudo tcpreplay -i eth0 -K --loop 1 smallFlows.pcap #here
```

Connection 4: Consume Kafka topic.

```
~/kafka/bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic instance-1-pcap --from-beginning
```

```
~/kafka/bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic instance-1-pcap
```

Make the instance 1 server publicly accessible and restrict access to a trusted list of source IP addresses using firewall rules. port 9092 open as TCP connection from any IP 0.0.0.0/0. Not secure. Better to restrict to a static [?] IP, but the default GCP IP is dynamic [?]. Thus, keeping the port open to any IP for demonstration purposes. It is advised to setup a shared VPC network. <https://cloud.google.com/vpc/docs/shared-vpc>

Compute power may have to be increased on instance-1, as tcpreplay maxes out the CPU (97%) even while repalying smallFlows.pcap (9444731B) versus Monday-WorkingHours.pcap (10822507416B) and Wednesday-WorkingHours.pcap (3420789612B)

Might assumed that not all packets are getting played or packets are delayed, buffered [?]

Or consider this instead, if Kafka in the way.

instance-1, session 1

```
sudo tcpdump -i eth0 -nn -vvvv | netcat [cluster-1-m's internal IP*] 4444
```

\*Was 10.128.0.4

instance-1, session 2

```
sudo tcpreplay -i eth0 -K --loop 1 smallFlows.pcap
```

Can listen on cluster-1-m and see the PCAP (optional)

```
netcat -l -p 4444
```

Or write it to a file

```
netcat -l -p 4444 > tmp.txt
```

Was able to wrap the instance-1 tcpreplay and pipe to netcat in `python3 server_threads.py` But first have master node listening `netcat -l -p 4444`

#here 2020-05-07 Have Spark consume stream with the normal socket receiver. Skipping Kafka for now.

First thing to do is concat every second line, but check the tcpdumps -v mode level etc Also review tcpreplay's settings. Is it replaying everything possible? Which also reminds me, tcpreplay needs better CPU.

## Packet analytics

---

The Storm or Spark cluster consumes the Kafka PCAP topic in order to parse, filter, reduce, analyze, and model packet capture.

Create cluster for Spark. Master and two workers:

- cluster-1-m
- cluster-1-w-0
- cluster-1-w-1

Create a cluster for Spark with Dataproc

<https://console.cloud.google.com/dataproc/clusters?project=user0112358d>

Cloud Dataproc API has been enabled

APIs Explorer Quickstart—Create a cluster "clusterName": "cluster-1", "clusterUuid": "b8ae0245-a98b-4b04-a388-2ab51a503b16",

1 master node, two workers

Worker nodes Each contains a YARN NodeManager and a HDFS DataNode.

see <https://www.quora.com/What-is-the-best-hardware-configuration-to-run-Hadoop>

Configure master

```
sudo apt-get install python3-pip
pip3 install pyspark
```

<https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html#quick-example>

```
/home/oberljn/.local/lib/python3.5/site-packages/pyspark/bin/spark-submit test_spark.py localhost 6969
```

See spark code test\_spark.py. Run Spark code like this:



```
/home/oberljn/.local/lib/python3.5/site-packages/pyspark/bin/spark-submit test_spark.py 10.150.0.6 4444
```

But first (or second...) on instance-1 run:

```
python3 server_threads.py
```

Spark code must include first off:

```
import os
x = '--packages org.apache.spark:spark-streaming-kafka-0-8_2.11:2.0.2 pyspark-shell'
os.environ['PYSPARK_SUBMIT_ARGS'] = x

from pyspark import SparkContext
from pyspark.streaming import StreamingContext
from pyspark.streaming.kafka import KafkaUtils

# Set up the Spark context and the streaming context
sc = SparkContext(appName="PCAP test")
sc.setLogLevel("WARN")

ssc = StreamingContext(sc, .01)

kafkaStream = KafkaUtils.createStream(ssc,
    '10.150.0.6:9092',
    'spark-streaming',
    {'instance-1-pcap':1})

kafkaStream.pprint()
#import re
#parsed = kafkaStream.map(lambda line: re.compile())

ssc.start()
sleep(5)
ssc.stop(stopSparkContext=True, stopGraceFully=True)
```

## to do: #here

- Get kafka connector to pyspark streaming df
- Do a transformation
- then run that pyspark file as a job in browser dataproc page

Packet extract namespace:

- number [?]
- time
- source
- destination
- protocol
- length
- flag

Create a file for parsing development and testing.

```
session 1 $ sudo tcpdump -i eth0 -nn -c 5 -vvvv > tmp.txt
session 2 $ sudo tcpreplay -i eth0 -tK --loop 3 smallFlows.pcap
```

See packet parse namespace and regular expressions at [packet\\_parse.json](#). See Python packet parse code at [packet\\_parse.py](#)

## Storage

---

PCAP data and KPIs from Storm processes are stored in a Cassandra database on Hadoop. Cassandra has been chosen due to the need for high availability in an environment like cyber ops. Cassandra has multiple master nodes that can continue to run the DB if one goes down. Whereas MongoDB has one master node in a cluster that, if it goes down, is not replaced until after 10 to 30 seconds. During the replacement process, the cluster cannot take input.

## Ad hoc reporting

---

Cassandra's query language CQL.

## Cyber ops dashboard

---

D3.js. Open source, web standards, mobile version

## References and resources

---

- Packet capture (pcap) is a performant C++ probe that captures network packets and streams them into Kafka. A pcap Storm topology then streams them into Cloudera Cybersecurity Platform (CCP)  
<https://docs.cloudera.com/ccp/2.0.1/add-new-telemetry-data-source/topics/ccp-pcap.html>
- Stream Processing vs. Continuous PCAP: The Big Shift in <https://www.extrahop.com/company/blog/2016/stream-processing-vs-continuous-pcap-the-big-shift-in-network-monitoring-architectures/>
- Radford. Network Traffic Anomaly Detection Using Recurrent Neural Networks. 28 Mar 2018. url: <https://arxiv.org/pdf/1803.10769.pdf>
- 
- Iman Sharafaldin, Arash Habibi Lashkari, and Ali A. Ghorbani. "Toward Generating a New Intrusion Detection Dataset and Intrusion Traffic Characterization". 4th International Conference on Information Systems Security and Privacy (ICISSP), Portugal, January 2018.
- <https://www.digitalocean.com/community/tutorials/how-to-install-apache-kafka-on-debian-9>
- <https://cloudwafer.com/blog/installing-apache-kafka-on-debian-9/>
- Python kafka
- <https://opensource.com/article/18/10/introduction-tcpdump>
- AppNeta <http://tcpreplay.appneta.com/wiki/tcpreplay-man.html>
- ref: Protocol Numbers <https://www.iana.org/assignments/protocol-numbers/protocol-numbers.xhtml>
- ref: Description of the Internet Protocol, IP <https://www.eit.lth.se/ppplab/IPHeader.htm>
- ref: pcaptools <https://github.com/caesar0301/awesome-pcaptools>
- <https://alvinalexander.com/linux-unix/linux-processor-cpu-memory-information-commands/>
- smallFlows.pcap <https://tcpreplay.appneta.com/wiki/captures.html>
- Datasets <http://205.174.165.80/CICDataset/CIC-IDS-2017/Dataset/> GeneratedLabelledFlows.zip This is already processed PCAPs and its resolution is only minutes.
- <http://205.174.165.80/CICDataset/CIC-IDS-2017/Dataset/PCAPs/> Friday-WorkingHours.pcap (<http://205.174.165.80/CICDataset/CIC-IDS-2017/Dataset/PCAPs/Friday-WorkingHours.pcap>)
- Protocol numbers <https://www.iana.org/assignments/protocol-numbers/protocol-numbers.xml>

## Useful snippets

---

System info.

```
top
cat /proc/cpuinfo
cat /proc/meminfo
free -m
```

Interface info.

```
ip link show
```

## Pyspark client

---

```
from pyspark import SparkConf, SparkContext
from pyspark.streaming import StreamingContext

def datastream():
    conf = SparkConf()
    conf.setAppName('TwitterStreamApp')

    sc = SparkContext(conf=conf)
    sc.setLogLevel('ERROR')

    ssc = StreamingContext(sc, 2) # interval size 2 seconds

    # setting a checkpoint to allow RDD recovery [?]
    ssc.checkpoint('checkpoint_TwitterApp')

    # read data from port 9009
    datastream = ssc.socketTextStream('localhost', 9009)

    return datastream
```

## Wrapping the traffic simulator and Kafka pub/sub

---

Saves from opening 4-5 sessions to instance-1. Code to run the traffic simulator and sniffer. Maybe separate these out, as they're different concepts.

Not tested

```
import subprocess
import threading
netcat_port = 8888
kafka_port = 9092
pcap_file = 'smallFlows.pcap'
def netcat_to_kafka(netcat_port, kafka_port):
    ...

    Pipe netcat to Kafka producer
    ...

    cmd = 'netcat -l -p {} | ~/kafka/bin/kafka-console-producer.sh \
--broker-list localhost:{} \
--topic instance-1-pcap > /dev/null'.format(netcat_port, kafka_port)
    subprocess.run(cmd)

def tcpdump_to_netcat(netcat_port):
    ...

    Sniff traffic on interface and pipe to netcat
    ...

    cmd = 'sudo tcpdump -i eth0 -nn -vvvv | \
netcat localhost -p {}'.format(netcat_port)
```

```
subprocess.run(cmd)

def tcpreplay(pcap_file):
    '''
    Simulate network traffic on the interface with tcpreplay
    '''

    cmd = 'sudo tcpreplay -i eth0 -K --loop 1 {}'.format(pcap_file)

    subprocess.run(cmd)
```

[^ top](#)

## Sources

---

- Iman Sharafaldin, Arash Habibi Lashkari, and Ali A. Ghorbani. Toward Generating a New Intrusion Detection Dataset and Intrusion Traffic Characterization. 4th International Conference on Information Systems Security and Privacy (ICISSP), Portugal, January 2018
- Resizing the file system and partitions on a zonal persistent disk. Google Cloud. url: [https://cloud.google.com/compute/docs/disks/add-persistent-disk?hl=en\\_US&\\_ga=2.94629659.-684521909.1584918365#resize\\_partitions](https://cloud.google.com/compute/docs/disks/add-persistent-disk?hl=en_US&_ga=2.94629659.-684521909.1584918365#resize_partitions)
- tcpreplay man page. Debian. url: <https://manpages.debian.org/unstable/tcpreplay/tcpreplay.1.en.html>

[^ top](#)