BGP Measurements Project – Summer 2024

Table of Contents

Motivation

In this assignment, we will explore Internet Measurements, a field of Computer Networks which focuses on large scale data collection systems and techniques that provide us with valuable insights and help us understand (and troubleshoot) how the Internet works. There are multiple systems and techniques that focus on DNS measurements, BGP measurements, topology measurements, etc. There are multiple conferences in this area, which we invite we to explore

and keep up with the papers that are published. The IMC conference is one of the flagship conferences in this area: ACM Internet Measurement Conference

A gentle introduction into the Internet Measurement field is to work with large scale BGP measurements and data to study topics such as:

- Characterizing growth of the Internet using various measures, such as number of advertised prefixes, the number of Autonomous Systems, the percentage growth of prefixes advertised by Autonomous System, and the dynamics of Autonomous System path lengths
- Inferring problems related to short-lived Announcement and Withdrawals,
- Inferring possible DDoS attacks by identifying community countermeasures such as "Remote Triggered Blackholing"

Introduction

In this project we will use the BGPStream tool and its Python interface PyBGPStream to understand the BGP protocol and interact with BGP data. The goal is to gain a better understanding of BGP and to experience how researchers, practitioners, and engineers have been using BGPStream to gain insight into the dynamics of the Internet. If we are interested in going deeper, we can use these same tools to observe and analyze real-time BGP data or download and analyze other historical BGP data.

Project Overview and Background

This project description, in combination with the comments in **`bgpm.py`**, comprise the complete requirements for the project. There are two complete sets of data included in the zip file and the provided test harness in **`check_solution.py`** will test each of the functions against both sets of data. We are welcome to copy and modify **`check_solution.py`** to better suit the development and debugging workflow, but we will have the best chance of success with the hidden data set used for grading if the final submission passes all the tests in the unmodified **`check_solution.py`**.

This project is designed to work in the class VM where the BGPStream libraries are installed. The code will need to run without modification in the course VM.

Some of the functions will have runtimes of several minutes. There is a lot of data to process, so the best way to speed up those functions is by focusing on the efficiency of the implementation. It is possible, but not supported, to install BGPStream and PyBGPStream on the local machine. Please don't ask TA staff for help if we decide to do this. **Gradescope imposes a hard time limit of 40 minutes for a grading session. We have no control over this and will not be able to make any allowances if the submission does not complete within that time limit.**

Required Background

For this project, we will be using BGPStream, an *open-source software framework for live and historical BGP data analysis, supporting scientific research, operational monitoring, and postevent analysis*. BGPStream and PyBGPStream are maintained by the Center for Applied Internet Data Analysis (CAIDA).

Read the resources

A high-level overview about how the BGPStream tool was developed was published by CAIDA in *BGPStream: A Software Framework for Live and Historical BGP Data Analysis*. This paper provides useful background and practical examples using BGPStream, so be sure to read it. Additionally, we should read *African peering connectivity revealed via BGP route collectors*, which provides a practical illustration of how the BGP collection system works.

Run Example Code Snippets

All the tasks are to be implemented using the Python interface to BGPStream. We are strongly encouraged to browse the following resources to familiarize yourself with the tool, and to run the example code snippets:

- PyBGPStream API: https://bgpstream.caida.org/docs/api/pybgpstream

- PyBGPStream API Tutorial: https://bgpstream.caida.org/docs/tutorials/pybgpstream
- PyBGPStream Repository: https://github.com/CAIDA/pybgpstream
- Official Examples: https://github.com/CAIDA/pybgpstream/tree/master/examples

Important Note

As will become apparent when we peruse the above documentation and tutorial information, the majority of BGPStream use cases involve gathering data – either live or historical – directly from the Route Collectors (which we refer to simply as "collectors"). The code for accessing a collector or set of collectors directly usually looks like this:

```
stream =
pybgpstream.BGPStream(     record_type="updates",
from_time="2017-07-07 00:00:00",     until_time="2017-
07-07 00:10:00 UTC",     collectors=["route-views.sg",
"route-views.eqix"],     filter="peer 11666 and prefix
more 210.180.0.0/16"
)
```

Each of the parameters to **`pybgpstream.BGPStream()`** winnows the data retrieved from the collector(s). **Because we are using pre-cached historical data in this project, we will not need to specify a collector or a time range. We also don't need to use any additional filtering.**

For this project, we can use set up and configure the streams with:

```
stream = pybgpstream.BGPStream(data_interface="singlefile")
stream.set_data_interface_option("singlefile", type, fpath)
```

where **`type`** is one of **`["rib-file", "upd-file"]`**[1] and **`fpath`** is a string representing the path to a specific cache file. When processing multiple files, we will create one stream per file.

Familiarize Yourself with the BGP Record Format and BGP Attributes

It is critical that we understand the BGP record format, especially the meaning and content of the fields (data attributes). A detailed explanation of BGP records and attributes can be found in RFC 4271: A Border Gateway Protocol 4 (BGP-4).

[1] We can see a complete list of types by running: **`bgpreader --data-interface singlefile -o?`**

It's also worth spending some time exploring the provided data using the BGPReader command line tool ("*a command line tool that prints to standard output information about the BGP records and the BGP elems that are part of a BGP stream*"). Doing so will be particularly helpful in understanding how the fields described in RFC 4271 and elsewhere map to the *BGP record* and *BGP elem* concepts used by BGPStream and PyBGPStream.

Because PyBGPStream allows we to extract the BGP attributes from BGP records using code, we ***will not*** have to interact with the BGP records in this format, but it is, nevertheless, helpful to see some examples using BGPReader to understand the fields. The next section shows

Here, we will show sample command line output from BGPReader for ***illustration*** purposes:

```
# read records from an update file, filtering for IPv4 only bgpreader
-e --data-interface singlefile --data-interface-option \ upd-
file=./rrc04/update_files/ris.rrc04.updates.1609476900.300.cache \
--filter 'ipv 4'

# read records from a rib file, filtering for IPv4 only bgpreader
-e --data-interface singlefile --data-interface-option \ rib-
file=./rrc04/rib_files/ris.rrc04.ribs.1262332740.120.cache \ --
filter 'ipv 4'
```

Update Example

The box below contains an example of an update record. In the record, the "|" character separates different fields. In yellow we have highlighted the **type** (A stands for Advertisement), the **advertised prefix** (210.180.224.0/19), **the path** (11666 3356 3786), and the **origin AS** (3786).

```
update|A|1499385779.000000|routeviews|routeviews.eqix|None|None|11666|206.
126.236.24|11666 3356 3786|11666:1000 3356:3 3356:2003 3356:575
3786:0 3356:22 11666:1002 3356:666 3356:86|None|None
```

RIB Example

The following is a Routing Information Base (RIB) record example. Consecutive "|" characters indicate fields without data.

```
R|R|1445306400.000000|routeviews|route-
views.sfmix|||32354|206.197.187.5|1.0.0.0/24|206.197.187.5|3235
4 15169|15169|||
```

Setup

Do not rely on the directory layout of the provided data. Gradescope does not mirror the directory layout from the provided files. Specifically, in the final submission, do not directly access the filesystem in any way and do not import all or part of either **`os`** or **`pathlib`**. All filesystem interaction will occur via PyBGPStream and the file paths will be taken from the Python list in the parameter named **`cache_files`** that is passed to each function.

Cache Files / Snapshots

Locate the directory **`rrc04/rib_files`** included with this assignment. This directory contains RIB dump files. Each filename (e.g., **`ris.rrc04.ribs.1262332740.120.cache`**) includes the collector's name (**`ris.rrc04`**), the type of data (**`ribs`**), and the Unix timestamp of the data (**`1262332740`**, which we can convert to a date via either of the two above links).

Each of the cache files is a snapshot of BGPM data collected by the collector at the time of the timestamp. In the rest of this assignment the term "snapshot" refers to the data in a particular cache file. ***Do not pull the own data***. The solution will be graded using cached data only.

We will need to write code to process the cache files. Each entry in **`cache_files`** is a string containing the full path to a cache file. To access a given path, the code will need to set up the appropriate data interface in the **`BGPStream()`** constructor:

```
stream = pybgpstream.BGPStream(data_interface="singlefile")
stream.set_data_interface_option("singlefile", type, fpath)
```

where **`type`** is one of **`["rib-file", "upd-file"]`** and **`fpath`** is a string representing the path to a specific cache file. When processing multiple files, we will create one stream per file.

Task 1. Understanding BGP Routing table Growth

In this task we will measure the growth over time of Autonomous Systems and advertised prefixes. The growth of unique prefixes contributes to ever-growing routing tables handled by routers in the Internet core. As optional background reading, please read the seminal paper *On Characterizing BGP Routing Table Growth*.

Task 1A: Unique Advertised Prefixes Over Time

This task will use cache files from the **`rib_files`** subdirectories. These are RIB files, so we will pass "rib-file" in the call to **`set_data_interface_option()`**. Using the data from cache files, measure the number of **unique advertised prefixes over time**. Each file is an annual snapshot. Calculate the number of unique prefixes within each snapshot by completing the function **`unique_prefixes_by_snapshot()`**. Make sure that the function returns the data structure exactly as specified in **`bgpm.py`**.

Task 1B: Unique Autonomous Systems Over Time

This task will use cache files from the **`rib_files`** subdirectories. These are RIB files, so we will pass "rib-file" in the call to **`set_data_interface_option()`**. Using the data from the cache files, measure the number of **unique Autonomous Systems over time**. Each file is an annual snapshot. Calculate the number of unique ASes within each snapshot by completing the function **`unique_ases_by_snapshot()`**. Make sure that the function returns the data structure exactly as specified in **`bgpm.py`**.

Note: Consider all paths in each snapshot. Here, we consider all AS that appear in the paths (not only the origin AS). We may encounter corner cases of paths with the following form: "`25152 2914 18687 {7829,14265}`". In this case, consider the AS in the brackets as a single AS. So, in this example, we will count 4 distinct ASes.

Task 1C: Top-10 Origin AS by Prefix Growth

This task will use cache files from the **`rib_files`** subdirectories. These are RIB files, so we will pass "rib-file" in the call to **`set_data_interface_option()`**. Using the data from the cache files, calculate the percentage growth in advertised prefixes for each AS over the entire timespan represented by the snapshots by completing the function **`top_10_ases_by_prefix_growth()`**. Make sure that the function returns the data structure exactly as specified in **`bgpm.py`**.

Consider each origin AS separately and measure the growth of the total unique prefixes advertised by that AS over time. To compute this, for each origin AS:

1. Identify the first and the last snapshot where the origin AS appeared in the dataset.
2. Calculate the percentage increase of the advertised prefixes, using the first and the last snapshots.
3. Report the top 10 origin AS sorted smallest to largest according to this metric. In the event of a tie (i.e., the same percentage increase), the AS with the lower number should come first.

Corner case: When calculating the prefixes originating from an origin AS, we may encounter paths of the following form: "`25152 2914 18687 {7829,14265}`". This is a corner case, and it should affect only a small number of prefixes. In this case, we consider the entire set of AS "`{7829,14265}`" as the origin AS.

Task 2: Routing Table Growth: AS-Path Length Evolution Over Time

In this task we will measure if an AS is reachable over longer or shorter path lengths as time progresses. Towards this goal we will measure the AS path lengths, and how they evolve over time. This task will use cache files from the **`rib_files`** subdirectories. These are RIB files, so we will pass "rib-file" in the call to **`set_data_interface_option()`**. Using the data from the cache files, calculate the shortest path for each origin AS in each snapshot by completing the function

`shortest_path_by_origin_by_snapshot()`. Make sure that the function returns the data structure exactly as specified in **`bgpm.py`**.

For each snapshot, we will compute the shortest AS path length for each origin AS in the snapshot by following the steps below:

- Identify each origin AS present in the snapshot. For example, given the path "11666 3356 3786", "3786" is the origin AS.
- For each origin AS, identify all the paths for which it appears as the origin AS.
- Compute the length of each path by considering each AS in the path only once. In other words, we want to remove the duplicate entries for the same AS in the same path and count the total number of unique AS in the path.
- ***Example*:** Given the path "25152 2914 3786 2914 18313", "18313" is the origin AS and "2914" appears twice in the path. This is a path of length 4.
- Among all the paths for an AS within the snapshot, compute the shortest path length.
- Filter out all paths of length 1.
- ***Corner cases*:** The data that we are working with are real data, which means that there may be ***few*** corner cases. In the vast majority of cases, paths have a straightforward form of "25152 2914 3786", but we might encounter corner cases such as:
    a. If an AS path has a single unique AS or a single repeated AS (e.g., "25152 25152 25152"), the path has length 1 and should be ignored
    b. An AS path entry that looks like "{2914,14265}" is an aggregate or AS_SET and constitutes a single AS path entry. It does not need to be parsed in any way. We can read more about aggregation in RFC 4271.
    Example: The length of the AS path "25152 2914 18687 {2914,14265} 2945 18699" is 6.
    Example: The length of the AS path "25152 2914 18687 18687 {18687}" is 4. The entries "18687" and "{18687}" are distinct, so we only deduplicate "18687".
    c. We can ignore all other corner cases.

Task 3: Announcement-Withdrawal Event Durations

In this task, we will measure how long prefix Announcements last before they are withdrawn. This matters because, when a prefix gets Advertised and then Withdrawn, this information propagates and affects the volume of the associated BGP traffic. Optional background reading on this topic can be found in *The Shape of a BGP Update*.

This task will use cache files from the **`update_files`** subdirectories. These are update files, so we will pass “upd-file" in the call to **`set_data_interface_option()`**. Using the data from the cache files, we will measure how long prefix Announcements last before they are withdrawn by completing the function **`aw_event_durations()`**. Make sure that the function returns the data structure exactly as specified in **`bgpm.py`**.

In defining Announcement Withdrawal (AW) events, we will only consider ***explicit withdrawals***. An explicit withdrawal occurs when a prefix is advertised with an (**A**)nnouncement and is then (**W**)ithdrawn. In contrast, an implicit withdrawal occurs when a prefix is advertised (A) and then re-advertised (A) - usually with different BGP attributes.

To compute the duration of an Explicit AW event for a given peerIP/prefix, we will need to monitor the stream of (**A**)nnouncements and (**W**)ithdrawals separately per peerIP/prefix pair.

- ***Example***: Given the stream: A1 A2 A3 W1 W2 W3 W4 for a specific peerIP/prefix pair, we have an implicit withdrawal A1-A2, another implicit withdrawal A2-A3, and, finally, an **explicit withdrawal (and AW event) A3-W1**. W1-W2, W2-W3, and W3-W4 are all meaningless, as there’s no active advertisement. The duration of the AW event is the time difference between A3 and W1. Again, we are only looking for **last A and first W.**
- ***Example***: Given the stream: A1 A2 A3 W1 W2 W3 W4 A4 A5 W4 for a specific peerIP/prefix pair, we have two AW events at A3-W1 and A5-W4.
- We consider only non-zero AW durations.

Task 4: RTBH Event Durations

In this task we will identify and measure the duration of Real-Time Blackholing (RTBH) events.

We will need to become familiar with Blackholing events. Good resources for this include *RFC 7999, Section 2*, *BGP communities: A weapon for the Internet (Part 2)*, and the video *Nokia - SROS: RTBH - Blackhole Community*.

This task will use cache files from the **`update_files_blackholing`** subdirectories. These are update files, so we will pass “upd-file" in the call to **`set_data_interface_option()`**. Using the data from the cache files, we will identify events where prefixes are tagged with a Remote Triggered Blackholing (RTBH) community and measure the time duration of the RTBH events by completing the function **`rtbh_event_durations()`**. Make sure that the function returns the data structure exactly as specified in **`bgpm.py`**.

The duration of an RTBH event for a given peerIP/prefix pair is the time elapsed between the last (**A**)nnouncement of the peerIP/prefix that is tagged with an RTBH community value and the first (**W**)ithdrawal of the peerIP/prefix. In other words, we are looking at the stream of Announcements and Withdrawals for a given peerIP/prefix and identifying only **explicit withdrawals** for an RTBH tagged peerIP/prefix.

To identify and compute the duration of an RTBH event for a given peerIP/prefix, we will need to monitor the stream of (**A**)nnouncements and (**W**)ithdrawals separately per peerIP/prefix pair.

- ***Example***: Given the stream: A1 A2 A3(RTBH) A4(RTBH) W1 W2 W3 W4 for a specific peerIP/prefix pair, A4(RTBH)-W1 denotes an RTBH event and the duration is calculated by taking the time difference between A4(RTBH) and W1.
- ***Note***: There can be more than one RTBH event in a given stream. For example, in the stream A1 A2 A3(RTBH) A4(RTBH) W1 W2 W3 W4 A5(RTBH) W5, there are two RTBH events: A4(RTBH)-W1 and A5(RTBH)-W5.

- ***Example***: Given the stream A1 A2 A3(RTBH) A4 A5 W1 W2 for a specific peerIP/prefix pair, the announcement A3(RTBH) followed by A4 is an implicit withdrawal. There is no explicit withdrawal and, thus, no RTBH event.
- In case of duplicate announcements, use the latest.
- Consider only non-zero duration events.