BGPStream: a framework for live and historical BGP data analysis

Chiara Orsini, Alistair King, Alberto Dainotti
CAIDA, UC San Diego

ABSTRACT

We present the design and implementation of BGPStream, an open-source software framework for the analysis of both historical and real-time Border Gateway Protocol (BGP) measurement data. Although BGP is a crucial operational component of the Internet infrastructure, and is the subject of research in the areas of Internet performance, security, topology, protocols, economics, etc., there is no standard and efficient way of processing large amounts of distributed and/or live BGP measurement data. BGPStream fills this gap, enabling efficient investigation of events, rapid prototyping, and building complex tools and efficient large-scale monitoring applications (e.g., detection of connectivity disruptions or BGP hijacking attacks). We discuss the design choices and challenges in the development of BGPStream. We present how the components of the framework can be used in different applicative scenarios, and we describe the development and deployment of complex services for global Internet monitoring that we built on top of it.

1. INTRODUCTION

We present the design and implementation of BGPStream, an open-source software framework (available at `bgpstream.caida.org`) for the analysis of historical and live Border Gateway Protocol (BGP) measurement data.

Although BGP is a crucial operational component of the Internet infrastructure, and is the subject of fundamental research (in the areas of performance, security, topology, protocols, economy, etc.), there is no standard and easy way of processing large amounts of BGP measurement data. BGPStream fills this gap by making available a set of APIs and tools for processing large amounts of live and historical data, thus supporting investigation of specific events, rapid prototyping, and building complex tools and efficient large-scale monitoring applications (e.g., detection of connectivity disruptions or BGP hijacking attacks). We describe the design choices and challenges in the development of BGPStream. We present how the components of the framework can be used in different applicative scenarios, and we describe the development and deployment of complex services for global Internet monitoring that we built on top of it.

2. BACKGROUND

The Border Gateway Protocol (BGP) is the de-facto standard inter-domain routing protocol for the Internet: its primary function is to exchange reachability information among Autonomous Systems (ASes) [36]. Each AS announces to the others, by means of BGP update messages, the routes to its local prefixes and the preferred routes learned from its neighbors. Such messages provide information about how a destination can be reached through an ordered list of AS hops, called an *AS path*.

A BGP router maintains this reachability information in the *Routing Information Base* (RIB) [36], which is structured in three sets:

- *Adj-RIBs-In*: routes learned from inbound update messages from its neighbors.
- *Loc-RIB*: routes selected from Adj-RIBs-In by applying local policies (e.g., shortest path, peering relationships with neighbors); the router will install these routes in its routing table to establish where to forward packets.
- *Adj-RIBs-Out*: routes selected from Loc-RIB, which the router will announce to its neighbors; for each neighbor the router creates a specific *Adj-RIB-Out* based on local policies (e.g., peering relationship).

Some operators make BGP routing information from their routers available for monitoring, troubleshooting and research purposes. BGP *looking glasses* give users limited (e.g., read-only) access to a command line interface of a router, or allow them to download the ASCII output of the current state of the router RIB. Looking glasses are more useful for interactive exploration rather than systematic and continuous data acquisition. The latter can instead be implemented either *(i)* by establishing a BGP peering session with the monitored router from a dedicated system (a *route collector*), or *(ii)* through a protocol specifically designed for monitoring purposes, such as OpenBMP [16, 41]. OpenBMP is

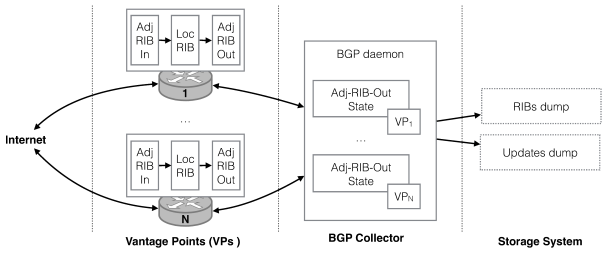

Figure 1: **BGP collection process illustrated.** Once a BGP collector establishes a BGP session with a VP, and it maintains a state and an image of the VP's Adj-RIB-out table derived from the updates received through the session. Periodically, it dumps a snapshot of the union of all the Adj-RIB-out tables (RIB dump) and the update messages received from all the VPs since the last dump (Updates dump).

an open-source implementation of the BGP Monitoring Protocol defined in an IETF draft [41] and supported by latest versions of JunOS and Cisco IOS. The protocol allows a user to periodically access the Adj-RIBs-In of a router or to monitor its BGP peering sessions. While OpenBMP can be easily deployed within an AS to monitor its BGP routers, there are currently no projects which make such data publicly available. Route collectors are often used for this purpose [32, 34, 39]. A route collector is a host running a collector process (e.g., Quagga [35]), which emulates a router and establishes BGP peering sessions with one or more real routers (*vantage points, VPs,* in the following). Each VP sends to the collector update messages (*updates*) each time the Adj-RIB-out changes, reflecting changes to its Loc-RIB (Figure 1).

Normally, a BGP session with a collector is configured as a *customer-provider* relationship, i.e., as if the VP was offering transit service to the collector. In this case, the VP is called *full-feed*, since it will advertise to the collector an Adj-RIB-Out which contains the entire set of routes in its Loc-RIB. This way, the collector potentially knows, at each instant, all the preferred-routes that the VP will use to reach the rest of the Internet — a partial view of the Internet topology graph visible to that router. A *partial-feed* VP instead, will provide through its Adj-RIB-Out only a subset of the routes in its Loc-RIB, e.g., routes to its own networks, or learned through its customers. Unfortunately, projects publicly providing information acquired by their collectors do not label VPs as full- or partial-feed, since peering with a collector is usually established on a voluntary basis and VP behavior can be subject to change without notice. Therefore, the policy that determines the Adj-RIB-Out to be shared with the collector must be dynamically inferred from the data (e.g., size of the Adj-RIB-Out).

For each VP, the collector maintains a session state and an image of the Adj-RIB-out table derived from updates. The collector periodically dumps, with a frequency of respectively few hours and few minutes, (i) a snapshot of the union of the maintained Adj-RIB-out tables (*RIB dump*) and (ii) the update messages received from all its VPs since the last dump, along with state changes (*Updates dump*).

The most popular projects operating route collectors and making their dumps available in public archives are RouteViews [32] and RIPE RIS [39]. They currently operate 18 and 13 collectors respectively, which in total peer with approximately 380 and 600 VPs distributed worldwide (this number increases every year). Analyzing data from multiple VPs is of fundamental importance for most Internet studies, since each router has a limited view of the Internet topology and, even when full-feed, a VP shares only part of this information (the preferred routes). Moreover, macroscopic Internet phenomena visible through the routing infrastructure (e.g., outages, cyber attacks, peering relationships, performance issues, route leaks, router bugs) affect Internet routers differently, as a function of geography, topology, router operating system and hardware characteristics, operator, etc.. RIB dumps provide an efficient summary of changes to BGP routing tables with a coarse time granularity that is sufficient for several classes of studies [20, 26–28]. In contrast, Updates dumps carry a lot of information to be processed, but offer a complete view of the observable routing dynamics, enabling other types of analysis and near-realtime monitoring applications [21, 22, 30, 44].

Such a distributed and detailed — even if partial – view of the inter-domain routing plane, generates large amounts of data. RouteViews and RIPE RIS collectors save a RIB dump every 2 and 8 hours and an Updates dump every 15 and 5 minutes, respectively. In 2015 an Adj-RIB-Out from a full-feed peer contains approximately 550k routes (each route includes an AS path toward a different network prefix) and on average generates about 1.5K updates every 5 minutes. Both projects save RIB and Updates dumps in a binary format, standardized by the IETF, called the Multi-Threaded Routing Toolkit (MRT) routing information export format [6]. The size of compressed dump files is currently between 10KB and 100MB for RIB dumps and between 1KB and 10MB for Updates dumps. RouteViews and RIPE RIS archives date back to 2001 and 1999 respectively, enabling longitudinal studies relevant to understand the evolution of the Internet infrastructure and its impact in other fields. The full archives of compressed files are about 8.9TB and 3.7TB, currently growing at the rate of 2TB per year.

The most widely adopted software for BGP data analysis in the research community [2, 4, 8, 23, 37, 40, 43] is *libBGPdump* [38], an open source C library that provides a simple API to parse BGP dumps in MRT format and deserializes MRT records into custom data struc-

tures. It is distributed along with a command-line tool, *bgpdump*, that outputs MRT information read from a file in an ASCII format. Often researchers directly use the command-line tool to translate entire BGP dumps into text, and then parse the ASCII output to further process or archive the data. Although bgpdump has been an invaluable tool to support the analysis of BGP data over the last decade, it lacks the advanced features that we discuss in the next section (e.g., merging and sorting data from multiple files and data sources, supporting live processing, scalability, etc.).

A solution that provides both retrieval simplicity and real-time access is *BGPmon* [1, 33, 45], a distributed monitoring system that retrieves BGP information by establishing BGP sessions with multiple ASes and that offers a live BGP data stream in the XML format (which also encapsulates the raw MRT data). Despite the fact that BGPmon enables rapid prototyping of live monitoring tools, it currently provides access to a limited number of VPs (compared to the vast number of VPs connected to RIS and RouteViews infrastructures), and it cannot be used for historical processing.

On the other hand, in the context of live monitoring, the major issue with popular public data sources such as RouteViews and RIPE RIS, is their file-based distribution system and thus the latency with which collected data is made available. Our measurements [14] show that, in addition to the 5 and 15 minutes delay due to file rotation duration, there is a small amount of variable delay due to publication infrastructure. However, 99% of Updates dumps in the last year were available in less than 20 minutes after the dump was begun. Since these latency values are low enough to enable several near-realtime monitoring applications, we began developing BGPStream with support for these data sources.

The research community recognizes the need for better support of live BGP measurement data collection and analysis. Since early 2015, we have been cooperating with other research groups and institutions (e.g., RouteViews, BGPMon, RIPE RIS) to coordinate efforts in this space [7]. Both RIPE RIS and BGPMon are developing a new BGP data streaming service (including support for streamed MRT records), and BGPMon partners with RouteViews to include in the forthcoming next-generation BGPMon service all of their collectors. Experience with the development of BGPStream informed development efforts of the other research teams and vice-versa. While BGPStream is fully usable today, we envision that the forthcoming developments of these projects, likely deployed in 2016, will enhance BGPStream capabilities.

3. DESIGN OVERVIEW

3.1 Goals

We designed the BGPStream software framework with the following goals:

– *Efficiently deal with large amounts of distributed BGP data.* In Section 2, we emphasized the importance of performing analyses by taking advantage of a large number of globally distributed vantage points.

– *Offer a time-ordered stream of data from heterogeneous sources.* BGPStream aims at providing a unified sorted stream of data from multiple collectors and collecting projects, i.e., interleaving into a single timeline the records they generate. Record-level sorting (rather than interleaving dump files) is important in at least two cases: (i) when analyzing long time intervals where it is infeasible to buffer the entirety of the input data in order to perform time alignment, and (ii) when at least one of the input data sources provides a continuous stream of data (rather than a discrete dump file), since such a stream cannot be interleaved at the dump file level.

– *Support near-realtime data processing.* BGP measurement data is fundamental to monitor the health of the global Internet. We support live monitoring applications consuming continuous streams of BGP data. We will consider two modes of operation: *(i)* **historical** - all the BGP data requested is available before the program starts; *(ii)* **live** - the BGP data requested becomes available while the program is running. In live mode, the time available for processing data is bounded; as such, the BGPStream stack of components, plus the user application, must process data faster than it is generated by VPs/collectors. Therefore, one of our objectives is to minimize processing latency caused by BGPStream, thus maximizing the processing time available to user applications.

Live mode also introduces the problem of sorting records from collectors that may publish data at variable times. Any solution to this problem, involves a trade-off between: (i) size of buffers, (ii) completeness of data available to the application, (iii) latency. Since such a trade-off should be evaluated depending on the specific goals and resources of the user application, we design BGPStream to perform best-effort record interleaving in live mode and we defer to the application the choice of a specific solution (in Section 7, we provide a concrete example of such a solution to support our infrastructure for live Internet monitoring).

– *Target a broad range of applications and users.* Potential applications of BGPStream are both in the field of network monitoring and troubleshooting as well as scientific data analysis. The target user base should not be limited to the availability of high-performance computing and/or cluster infrastructure. The BGPStream framework makes available a set of tools and APIs that suit different applications and development paradigms (e.g., historical data analysis, rapid prototyping, script-

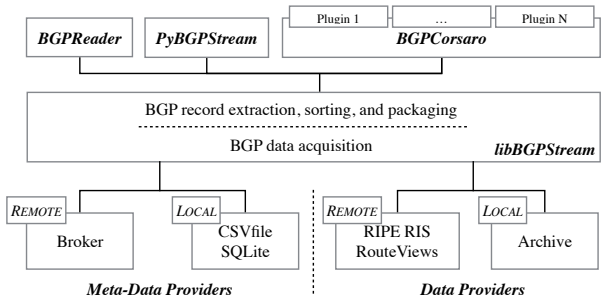

Figure 2: **BGPStream framework overview.** The BGPStream framework is organized in three layers. From bottom up, these are: data and meta-data access, records extraction and packaging (libBGPStream), and record processing (BGPReader, PyBGPStream, BGPCorsaro).

ing, live monitoring).

– *Scalability.* Since the pervasiveness of BGP VPs is key to monitoring and understanding the Internet infrastructure, the number of VP supported by collector projects continually grows. In parallel, the technological challenges (e.g., near-realtime detection of sophisticated man-in-the-middle attacks) require solutions of increasing complexity and computational demand. We designed BGPStream to enable deployment in distributed architectures (Section 7 shows an example use in a customized distributed environment). BGPStream is also suited for use in a "Big Data" environment: e.g., Spark's [3] native Python support makes BGPStream usable in such an environment out-of-the-box (Python bindings to the main BGPStream library are discussed in Section 6).

– *Easily extensible.* Though our solution is designed to work with current standards and the most popular available data sources, we designed the entire framework as a stacked and modular architecture, facilitating support for new technologies and data sources. BGPStream is indeed a project under evolution and is part of a coordinated effort with data providers, developers of complementary technologies, and users, to advance the state of the art in BGP monitoring and measurement data analysis [7].

In the next section we provide an overview of the main components of the BGPStream framework, whereas design and development choices are further discussed in Sections 4, 5, and 6.

3.2 Overview of Components

The BGPStream framework is organized in three layers: data and meta-data access, records extraction and packaging, and record processing (Figure 2).

1. The **data and meta-data access** layer provides to the upper layer information about BGP data availability as data annotations. One of the challenges in analyzing BGP measurement data is identifying and obtaining relevant data. Both Route Views and RIPE RIS make data available over HTTP, with basic directory-listing style indexes into the data. Identifying the appropriate files for large-scale analysis (across multiple collectors and long time durations) involves either manual browsing and download, or scripting of a crawler tailored to the structure of each project's repository. Downloading the data, may itself take a significant amount of time (e.g., all data collected in 2014 is ≈2TB). Moreover, since both projects continually add new data to their archives as it is collected (Section 2), near-realtime monitoring requires custom scripts to periodically scrape the websites and download new data. This layer hides all of these complexities through meta-data providers: components that provide access to information about the files hosted by local or remote data repositories (the *Data Providers*, e.g., the Route Views and RIPE RIS archives). (Section 4).

2. The **record extraction and packaging** layer is implemented by **libBGPStream**, the core library of the framework (Section 5), which provides the following functionalities:
   - Transparent access to concurrent dumps (i) from multiple collectors, (ii) of different collector projects, and (iii) of both RIB and Updates type.
   - Live data processing.
   - Data extraction, annotation and error checking.
   - Generation of a sorted (by timestamp) stream of BGP measurement data.
   - An API through which the user can specify and receive a stream.

3. The **record processing** layer consists of all the components that use libBGPStream's API. We distribute BGPStream with the following independent modules: **BGPReader**, a command-line tool that outputs the requested BGP data in ASCII format; **pyBGPStream**, Python bindings to the libBGPStream API; **BGPCorsaro**, a tool that uses a modular plugin architecture to extract statistics or aggregate data that are output at regular time bins. These components are compared in Figure 3 by contrasting their ease of use against computational efficiency.

4. META-DATA PROVIDERS

A meta-data provider is a component that provides access to information about the files hosted by data repositories (the *Data Providers*, e.g., the Route Views and RIPE RIS archives). In order to provide to BGPStream users a unified query interface to retrieve streams of data from different data providers, we designed a web

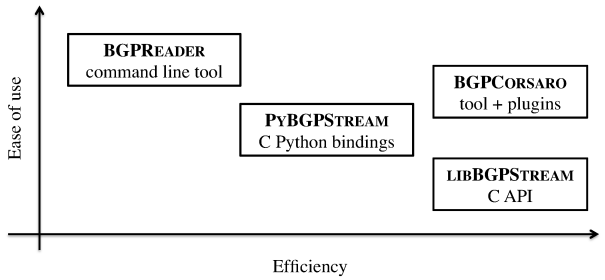

Figure 3: **BGPStream record processing toolkit**. Compares record process components by contrasting their ease of use (y axis) against computational efficiency (x axis).

service called *BGPStream Broker* which provides the following functionalities:

- Provide meta-data to libBGPStream
- Load balancing
- Response windowing for overload protection
- Support for live data processing

The Broker continuously scrapes data provider repositories, stores meta-data about every new file into an SQL database, and answers to HTTP queries to identify the location of files matching a set of parameters. An instance of the Broker is hosted at the San Diego Supercomputer Center at UC San Diego and is queried by default by a libBGPStream installation, allowing BGPStream to be used "out-of-the-box" on any Internet-connected machine. However, since we release the Broker as open source, an organization can deploy their own instance, potentially supporting custom (e.g., private) repositories.

The Broker stores only meta-data about files available on the official repository, not the files themselves. This approach minimizes the potential for a bottleneck since queries to, and responses from, the Broker are lightweight, with the actual data being served by dedicated data provider archives. This configuration also makes it simple to add support for additional data providers, as well as provide load-balancing and redundancy as the Broker can transparently round-robin amongst multiple mirror servers or adopt more sophisticated policies (e.g., requests sent from UC San Diego machines are normally pointed to campus mirrors). The meta-data collected includes: collector name (e.g., *route-views2*), data type (i.e., *rib* or *updates*), dump time, and dump duration.

The Broker is queried via an HTTP API that returns data in the JSON format. The API accepts as parameters, lists of collectors, dump types (*RIB dumps* and/or *Updates dumps*), and time intervals. The response contains a list of meta-data describing MRT files matching these parameters, sorted by time and type, and with the following parameters: `url`, the HTTP URL of the dump file; `project`, the collection project that owns the file; `collector`, the name of the collector that generated the dump; `type`, the type of MRT dump file (`RIB`, or `Updates`); `initialTime`, the nominal time that the dump was begun; `duration`, the number of seconds worth of data that the dump file contains (for RIB this is fixed to 120 seconds).

To prevent the Broker from being overwhelmed by large requests, each response will contain a window with at most 2 hours worth of dump files and the client will send additional queries to receive the subsequent windows. However, when processing data in live mode, we need to take into account the distributed and asynchronous nature of BGP data collection, which causes data from different collectors and projects to become available to the broker at different times. Therefore, it is possible that when queried for window W_i the broker has new data available for window $W_{(i-n)}$. For this reason, the client does not change the requested time interval between queries as it is used by the Broker to check previous windows for newly arrived data. Instead, to obtain the next window of data from the broker, the client repeats the original query setting two additional parameters:

- `minInitialTime`, which is used to define the start of the new window requested. The client sets it to the maximum of ($initialTime + duration$) for each file returned in previous query results. This effectively moves the new window to begin immediately following the previous result set.
- `dataAddedSince`, which is set to the value of the `time` attribute returned by the broker in the previous response (this is the timestamp of when the Broker queried the database). This is used by the broker to identify files added to the database since the last query.[1]

While the Broker Data Interface is the primary data access interface, we also provide three other interfaces for small-scale analysis of local files: *Single file*, *CSV file*, and *SQLite*. The single-file data interface allows at most one RIB and one Updates file to be provided directly to BGPStream, much like the legacy *BGPdump* application. The CSV file and SQLite data interfaces allow a local (private) MRT archive to be used with BGPStream. However, the CSV data interface is not suitable for live monitoring due to the need to lock the file when adding records, whereas the SQLite interface suffers degraded performance when the database holds meta-data for a large number of files. The following sections assume that the Broker is used as the Data Interface.

5. LIBBGPSTREAM

[1]This feature may be disabled by omitting the `dataAddedSince` parameter.

5.1 libBGPStream API

The libBGPStream user API provides the essential functions to configure and consume a stream of BGP measurement data and a systematic organization of the BGP information into data structures. The API defines a BGP data stream by the following parameters: *collector projects (e.g., Route Views, RIPE RIS), list of collectors, dump types* (RIB/Updates), *time interval start* and either *time interval end* or *live mode*. A stream can include dumps of different type and from different collector projects.

Listing 1, shows sample code that uses the BGPStream API to print out all the announcement and withdrawal messages for a specific prefix as observed by VPs connected to *rrc00* (a RIPE RIS collector) and *route-views2* (a Route Views collector) in the given time interval. Any program using the libBGPStream C API consists of a stream configuration phase and a stream reading phase: first, the user defines the meta-data filters (lines 15-19), then the iteratively requests new records to process from the stream (lines 25-42).

Listing 1, can be converted into a live monitoring process simply by setting the end of the time interval to *-1*.

5.2 Interface to Meta-Data and Data Providers

To access data and meta-data from the providers, the library implements a "client pull" model, which (i) enables efficient data retrieval without potential input buffer overflow (i.e., data is only retrieved when the user is ready to process it) and (ii) supports live mode.

To implement this model, the system iteratively alternates between making meta-data queries to the Broker (using the protocol described in Section 4), and opening and processing the dump files that are returned. When the Broker returns an empty dump file set, the system signals to the user that the stream has ended. In live mode however, the query mechanism is blocking: if the Broker has no data available, a polling cycle will begin, periodically re-issuing the request to the Broker until either the response from the Broker contains new files for processing, or libBGPStream receives an interrupt signal.

5.3 Data structures and error checking

libBGPStream requires BGP dump files to comply with the MRT format [6]. Dumps are composed of *MRT records*, whose type is specified in their header [6]. An update message is stored in a single MRT record, whereas a RIB dump is made of multiple MRT records. Specifically, a collector dumps in each MRT record composing a RIB dump, information related to a single prefix. The *BGPStream record* structure contains a deserialized MRT record, as well as an error flag, and additional annotations related to the originating dump

Listing 1 BGPstream prefix monitoring. An example program that uses the BGPStream API to print out all the announcement and withdrawal messages for a specific prefix as observed by VPs connected to *rrc00* and *route-views2*. To use the BGPStream API, programs first configure the stream (lines 15-19) and then iteratively request records from the stream (lines 25-42).

```
int main(int argc, const char **argv)                                                    1
{                                                                                        2
  bgpstream_t *bs = bgpstream_create();                                                  3
  bgpstream_record_t *record = bgpstream_record_create();                                4
  bgpstream_elem_t *elem = NULL;                                                         5
  char buffer[1024];                                                                     6
                                                                                         7
  /* Define the prefix to monitor for (2403:f600::/32) */                               8
  bgpstream_pfx_storage_t my_pfx;                                                        9
  my_pfx.address.version = BGPSTREAM_ADDR_VERSION_IPV6;                                  10
  inet_pton(BGPSTREAM_ADDR_VERSION_IPV6, "2403:f600::", &my_pfx.address.ipv6);           11
  my_pfx.mask_len = 32;                                                                  12
                                                                                         13
  /* Set metadata filters */                                                            14
  bgpstream_add_filter(bs, BGPSTREAM_FILTER_TYPE_COLLECTOR, "rrc00");                    15
  bgpstream_add_filter(bs, BGPSTREAM_FILTER_TYPE_COLLECTOR, "route-views2");             16
  bgpstream_add_filter(bs, BGPSTREAM_FILTER_TYPE_RECORD_TYPE, "updates");                17
  /* Time interval: 01:20:10 - 06:32:15 on Tue, 12 Aug 2014 UTC */                      18
  bgpstream_add_interval_filter(bs, 1407806410, 1407825135);                             19
                                                                                         20
  /* Start the stream */                                                                21
  bgpstream_start(bs);                                                                   22
                                                                                         23
  /* Read the stream of records */                                                      24
  while (bgpstream_get_next_record(bs, record) > 0) {                                    25
    /* Ignore invalid records */                                                        26
    if (record->status != BGPSTREAM_RECORD_STATUS_VALID_RECORD) {                        27
      continue;                                                                          28
    }                                                                                    29
    /* Extract elems from the current record */                                         30
    while ((elem = bgpstream_record_get_next_elem(record)) != NULL) {                    31
      /* Select only announcements and withdrawals, */                                  32
      /* and only elems that carry information for 2403:f600::/32 */                    33
      if ((elem->type == BGPSTREAM_ELEM_TYPE_ANNOUNCEMENT ||                             34
           elem->type == BGPSTREAM_ELEM_TYPE_WITHDRAWAL) &&                              35
          bgpstream_pfx_storage_equal(&my_pfx, &elem->prefix)) {                         36
        /* Print the BGP information */                                                  37
        bgpstream_elem_snprintf(buffer, 1024, elem);                                     38
        fprintf(stdout, "%s\n", buffer);                                                 39
      }                                                                                  40
    }                                                                                    41
  }                                                                                      42
                                                                                         43
  bgpstream_destroy(bs);                                                                 44
  bgpstream_record_destroy(record);                                                      45
  return 0;                                                                              46
}                                                                                        47
```

(Table 1).

To open MRT dumps, we use a version of libBGPdump [38] that we extended to: (i) read remote paths (HTTP and HTTPS), (ii) support opening and reading from multiple files in parallel from a single process, and (iii) signal the event of a corrupted read. libBGPStream uses the latter to set the *status* field in the BGPStream record to *not-valid* if the BGP dump file cannot be opened (e.g., the website that we are trying to access is temporarily down) or if the dump is corrupted (e.g., the MRT length in the header is not compatible with the size of the file). libBGPStream also marks records that *begin* or *end* a dump file, allowing users to collate records contained in a single RIB dump.

An MRT record (and therefore a BGPStream record) may group elements of the same type but related to different VPs or prefixes, such as routes to the same prefix from different VPs (in a RIB dump record), or

Table 1: **BGPStream record fields.**

Field	Type	Function
project	string	project name (e.g., Route Views)
collector	string	collector name (e.g., rrc00)
type	enum	RIB or Updates
dump time	long	time the containing dump was begun
position	enum	first, middle, or last record of a dump
time	long	timestamp of the MRT record
status	enum	record validity flag
MRT record	struct	de-serialized MRT record

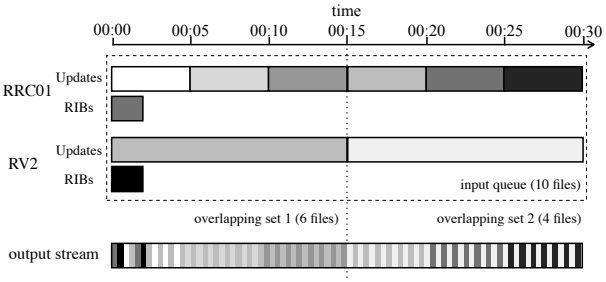

Figure 4: **Intra- and inter-collector sorting in libBGPStream.** An example showing how RIB and Updates dumps generated by a RIPE RIS collector (RRC01) and a Route Views collector (RV2) are interleaved into a sorted stream. The 30 minutes (10 files) of BGP data are first separated into two disjoint sets (of 6 and 4 files) based on overlapping file time intervals. Then a multi-way merge is applied separately to the two sets, yielding the stream depicted at the bottom.

announcements from the same VP, to multiple prefixes, but sharing a common path (in a Updates dump record). To provide access to individual elements, libBGPStream decomposes a record into a set of *BGPStream elem* structures (Table 2). We do not currently expose all the BGP attributes contained in a MRT record in the BGPStream elem; we will implement the remaining attributes in a future release.

Table 2: **BGPStream elem fields.**

Field	Type	Function
type	enum	route from a RIB dump, announcement, withdrawal, or state message
time	long	timestamp of MRT record
peer address	struct	IP address of the VP
peer ASN	long	AS number of the VP
prefix*	struct	IP prefix
next hop*	struct	IP address of the next hop
AS path*	struct	AS path
old state*	enum	FSM state (before the change)
new state*	enum	FSM state (after the change)

* denotes a field conditionally populated based on type

5.4 Generating a sorted stream

libBGPStream generates a stream of records sorted by the timestamps of the MRT records they encapsulate. Collectors write records in dump files with monotonically increasing timestamps. However, additional sorting is necessary when the stream is configured to include MRT records stored in files with overlapping time intervals[2], which occurs in two cases: (i) when reading dumps from more than one collector (inter-collector sorting); (ii) when a stream is configured to include both RIB and Updates dumps (intra-collector sorting). Since each file can be seen as an ordered queue of records, in practice, libBGPStream performs a *multi-way merge* [24].

To reduce the computational cost of sorting records, we perform multi-way merging separately on disjoint sets of files from the dump file queue (given the current number of collectors in Route Views and RIS, the dump files queue can contain up to ≈500 files). However, to ensure correct sorting, files with overlapping time intervals need to be in the same set. This problem is exacerbated by the fact that the duration of Updates dumps vary between projects.

We minimize the number of files per set by iteratively applying the following process until the queue is empty: *(1)* initialize a new set with the oldest file in the queue; *(2)* recursively add files with time intervals overlapping with at least one file already in the set; *(3)* remove the set of files from the queue. Such sets currently contain up to ≈150 files[3].

For each set, libBGPStream simultaneously opens all the files in the set and iteratively (i) extracts the oldest MRT record from such files, and (ii) uses the MRT record to populate a BGPStream record (Figure 4).

As noted in Section 3, sorting in live mode is best-effort and needs to be managed also by the user application. In Section 7.2, we provide an example of such a solution tailored to a specific live monitoring application.

6. RECORD PROCESSING

While users can write code that directly uses the services offered by the **BGPStream C API**, we distribute BGPStream with three solutions that will require writing much less (or no) code and fit a variegate set of applications.

6.1 ASCII command-line tool

BGPReader is a tool to output in ASCII format the BGPStream records and elems matching a set of filters given via command-line options. This tool is meant to support exploratory or ad-hoc analysis using command line and scripting tools for parsing ASCII data.

BGPReader can be thought of as a drop-in replacement of the analogous bgpdump tool (a command line

[2]We define the time interval associated with a dump file as the time range covered by the timestamps of its records.

[3]We also use this set creation algorithm in the Broker to ensure that files with overlapping intervals are returned in a single window. Since the overall time interval of a set of overlapping files is normally either 15 or 30 minutes, a 2 hour window will commonly contain approximately 8-16 file sets.

Listing 2 pyBGPstream AS path comparison.

```
from _pybgpstream import BGPStream, BGPRecord, BGPElem
from collections import defaultdict
from itertools import groupby
import networkx as nx

stream = BGPStream()
as_graph = nx.Graph()
rec = BGPRecord()
bgp_lens = defaultdict(lambda: defaultdict(lambda: None))
stream.add_filter('record-type','ribs')
stream.add_interval_filter(1438415400,1438416600)
stream.start()

while(stream.get_next_record(rec)):
    elem = rec.get_next_elem()
    while(elem):
        monitor = str(elem.peer_asn)
        hops = [k for k, g in groupby(elem.fields['as-path'].split(" "))]
        if len(hops) > 1 and hops[0] == monitor:
            origin = hops[-1]
            for i in range(0,len(hops)-1):
                as_graph.add_edge(hops[i],hops[i+1])
            bgp_lens[monitor][origin] = \
                min(filter(bool,[bgp_lens[monitor][origin],len(hops)]))
        elem = rec.get_next_elem()
for monitor in bgp_lens:
    for origin in bgp_lens[monitor]:
        nxlen = len(nx.shortest_path(as_graph, monitor, origin))
        print monitor, origin, bgp_lens[monitor][origin], nxlen
```

option sets bgpdump output format), which is widely used by researchers and practitioners. However, BGPReader adds features such as the support to read data from multiple files, collectors, and projects in a single process and to configure filters. Additionally, due to the parallelized reading of dump files provided by libBGPStream, processing multiple files is faster compared to bgpdump: for example, BGPReader processes 24 hours of data (August 15 2015), from 18 Route Views and 13 RIPE RIS collectors, in 156 minutes, whereas bgpdump takes 202 minutes (a 23% improvement).

6.2 Python bindings

pyBGPStream is a Python package that exports all the functions and data structures provided by the libBGPStream C API. We bind directly to the C API instead of implementing the BGPStream functions in Python, in order to leverage both the flexibility of the Python language (and the large set of libraries and packages available) as well as the performance of the underlying C library.

Even if an application implemented in Python using pyBGPStream would not achieve the same performance as an equivalent C implementation, pyBGPStream is an effective solution for: rapid prototyping, implementing programs that are not computationally demanding, or programs that are meant to be run offline (i.e., there are no time constraints associated with a live stream of data).

In Listing 2, we show a practical example related to a research topic commonly studied in literature: the AS path inflation [19, 42]. The problem consists in quantifying the extent to which routing policies inflate the

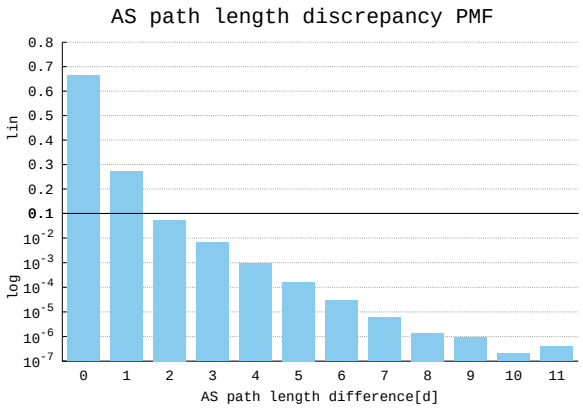

Figure 5: **The extent of AS paths inflation.** Probability mass function of the difference in length between the shortest AS path length observed in BGP and in the undirected graph for the same <monitor,origin> pairs.

AS paths (i.e., how many AS paths are longer than the shortest path between two ASes due to the adoption of routing policies), and it has practical implications, as the phenomenon directly correlates to the increase in BGP convergence time [25]. In less than 30 lines of code, the program compares the AS-path length observed in a set of BGP RIB dumps and the corresponding shortest path computed on a simple undirected graph built using the AS adjacencies observed in the AS paths. The program reads the 8am RIB dumps provided by all RIS and Route Views collectors on August 1st 2015, and extracts the minimum AS-path length observed between a monitor and each origin AS. While reading the RIB dumps, the program also maintains the AS adjacencies observed in the AS path. We then use the NetworkX package [31] to build a simple undirected graph (i.e., a graph with no loops, where links are not directed) and we compute the shortest path between the same <monitor,origin> AS pairs observed in the RIB dumps. Figure 5 compares path lengths of 10M unique <monitor,origin> AS pairs and shows that, in 30% of cases, inflation of the path between the monitor and the origin AS accounts for 1 to 11 hops.

6.3 Continuous monitoring using C plugins

BGPCorsaro is a tool to continuously extract derived data from a BGP stream in regular time bins. Its architecture is based on a pipeline of plugins, which continuously process BGPStream records. Plugins can be either:

- Stateless: e.g., performing classification and tagging of BGP records; plugins following in the pipeline can use such tags to inform their processing.
- Stateful: e.g., extracting statistics or aggregating data that are output at the end of each time bin. Since libBGPStream provides a sorted stream of records, BGPCorsaro can easily recognize the end

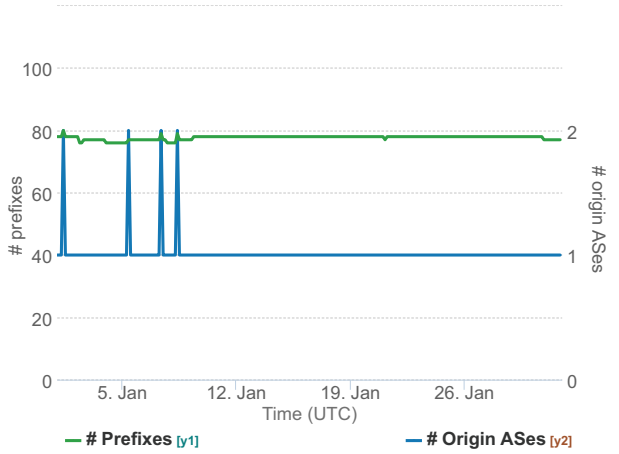

Figure 6: **Monitoring of GARR (AS137) IP space using the pfxmonitor plugin.** The green line reports the number of unique prefixes announced over time, the blue line reports the number of unique origin ASes that are currently announcing such prefixes. The spikes of the origin AS timeseries identify four hijack events in which AS 198596 announces part of the IP space belonging to AS137.

of a time bin even when processing data from multiple collectors.

Both the core and the plugins of BGPCorsaro are written in C in order to support high-speed analysis of historical or live data streams. In Section 7, we describe a deployment of BGPCorsaro that runs 24/7 as a part of our global Internet monitoring infrastructure.

As a sample plugin, we describe a stateful plugin that monitors prefixes overlapping with a given set of IP address ranges. For each BGPStream record, the plugin: (1) selects only the RIB and Updates dump records related to prefixes that overlap with the given IP address ranges. (2) tracks, for each <prefix, VP> pair, the ASN that originated the route to the prefix. At the end of each time bin, the plugin outputs the timestamp of the current bin, the number of unique prefixes identified and, the number of unique origin ASNs observed by all the VPs.

We used this plugin to process data from all available Route Views and RIPE RIS collectors, for January 2015, setting the time bin size to 5 minutes, and providing as input to the plugin the IP ranges covered by the 78 prefixes originated by AS137 (GARR, the Italian Academic and Research Network) as observed on January 1st, 2015. Figure 6, shows a graphical representation of the two time-series generated by the plugin: the number of unique announced prefixes (in green) and number of unique origin ASNs (in blue). While a small oscillation of the number of prefixes announced is expected (as prefixes can be announced as aggregated or de-aggregated), in 4 cases the number of unique announcing ASes shifts from 1 to 2, for about 1 hour. Through manual analysis, we found that, during these spikes, a portion of GARR's IP space (specifically, 7 /24 prefixes) was also announced by TehnoGrup (AS 198596), a Romanian AS that appears to have no relationship with GARR. The event on January 7th is reported as an hijack attack by Dyn Research [29], and given the similar nature of the other three events visible in the graph (1st, 7th and 8th of January), the plugin output suggests that three additional attacks occurred. Although this approach cannot detect all types of hijacking attacks, it is still a valid method to identify suspicious events and serves to demonstrate the capabilities of BGPCorsaro.

7. MONITORING THE GLOBAL INTERNET

In this section, we describe how we use BGPStream to develop and deploy our global BGP monitoring infrastructure supporting research into macroscopic Internet events. The purpose of this section is *(i)* to highlight how BGPStream enables the development of a complex monitoring system with stringent requirements, and *(ii)* to exemplify how additional challenges that arise in such complex BGP monitoring tasks — and which we do not address by-design in BGPStream – can be solved.

In the IODA research project [13], we constantly monitor the Internet to detect and characterize phenomena of macroscopic connectivity disruption [11] [12]. We combine information from different types of measurement, such as active probing, passive traffic analysis, and BGP data. In the case of BGP, our objective is to understand whether a set of prefixes (that, e.g., share the same geographical region, or the same origin AS) are globally reachable or not. Information from a single VP is not sufficient to verify the occurrence of an outage, in fact, a prefix may be not reachable from the VP because of a local routing failure. On the other hand, if several VPs, topologically and geographically dispersed, simultaneously lose visibility of a prefix, then it is very likely that the prefix itself is undergoing an outage.

Another class of events that we are interested in detecting and analyzing is BGP-based traffic hijacking [10]. The most common hijacks manifest as two or more distinct ASes announcing exactly the same prefix, or a portion of the same address space, at the same time. In order to detect such events, it is essential to compare the prefix reachability information as observed from multiple VPs. For example, in the case of a hijack with a man-in-the-middle attack [9], the Internet can be divided into two parts: one polluted by the illegitimate announcement, and one that still maintains the legitimate path towards the destination prefix.

Therefore, in order to monitor the Internet for these events in a timely fashion, we need to maintain a global (i.e., for each and every VP) view of BGP reachability information updated with fine time granularity (e.g., few minutes). In general, a continuously updated global

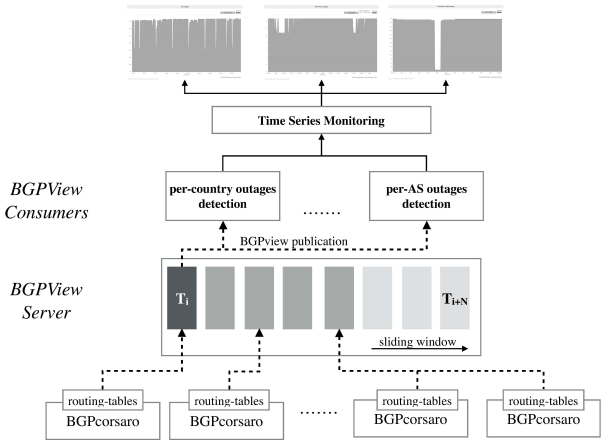

Figure 7: **BGPStream framework deployment for live monitoring**. For each collector, we run we run an instance of BGPCorsaro that runs the routing-tables plugin to maintain, in hash table, the observable LocRIB of all the VPs of the collector. At the end of each time bin (e.g., 1 minute) each BGPCorsaro pushes its hash table to the BGPViewServer that, in turn, aggregates hash tables with the same timestamp into partial BGP views and publishes them once they become complete. Such data can be further processed by the BGPViewConsumers, that then can handle the results of their computation to a Time Series Monitoring system.

view can be useful in many other applicative scenarios, such as tracking AS paths containing a particular AS, verifying the occurrence of a route leak, spotting new (suspicious) AS links appearing in the AS-graph, etc.

We implement our live monitoring system using the distributed architecture sketched in Figure 7. On top of BGPStream and BGPCorsaro, we implement three mechanisms:

- A solution to efficiently and accurately reconstruct the observable LocRIB of each VP (as discussed in Section 2, the LocRIB is fully observable only from full-feed peers; for simplicity, in this section we will refer to the observable LocRIB generically as the *routing table* of the VP): we developed a BGPCorsaro plugin, called *routing-tables*, that performs this operation at regular intervals of 1 minute (Section 7.1). We run one BGPCorsaro instance per collector in order to distribute the computation across multiple CPU-cores and/or hosts (the current prototype system runs on 2 machines, each with 12 CPU cores). Each BGPCorsaro instance pushes data to a system called *BGPViewServer* via a message queue.
- A synchronization mechanism that — in live mode – aligns data published with variable timing by multiple collectors: through a synchronization buffer, the BGPViewServer merges into *BGP views* the output from BGPCorsaro instances as it becomes available and publishes on average one BGP view every minute (Section 7.2).

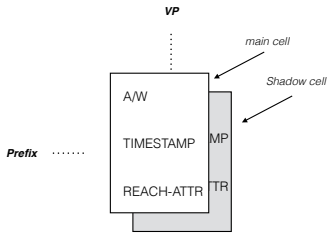

Figure 8: **A cell of the multi-dimensional hash table used by the routing-table BGPCorsaro plugin to maintain the state of a prefix for a VP.** The cell carries the prefix reachability attributes as observed by a VP and is updated by announcements, withdrawals, and RIB dump records.

- Analysis modules that implement data manipulation routines (e.g., for event detection or extraction of statistics to output as time series) on a BGP view, which we call BGPViewConsumers (Section 7.3). The communication between the BGPViewServer and the consumers follows a publish-subscribe model.

7.1 Reconstructing VPs routing tables

Since RIB dumps are currently dumped every 2 or 8 hours by Route Views and RIPE RIS, the routing-tables plugin uses a RIB dump as a starting reference and then relies on the Update dumps to reconstruct the evolution of the routing table, using subsequent RIB dumps for sanity checking and correction.

We save state and routing table information in a data structure organized as a multi-dimensional hash table, which provides insertion and lookup with average time complexity of $\mathcal{O}(1)$ and exploits the data redundancy of BGP routing tables from multiple VPs to reduce its memory footprint. At a high level, this structure is a matrix with prefixes and VPs as row and column indexes, respectively. Each cell in the matrix (Figure 8) contains the **reachability-attributes** for the prefix (e.g, the AS path), the **timestamp** of when the cell was last modified by an Update dump record, a **A/W** flag that indicates whether such operation was an announcement or a withdrawal, and a **shadow cell**, a similar structure except for the absence of the A/W flag.

The shadow cell is used to store data from a new RIB dump record before it is applied: we apply all the records from a RIB dump only if none of them is marked as corrupted by BGPStream. A RIB dump is uniquely identified by the BGPStream record fields <*project, collector, type, dump time*>, and the plugin recognizes its last record through the *position* field. Each time the last record of a (not corrupted) RIB dump is received, the information in the shadow cells in the columns (VPs) associated with the corresponding collector is compared to their respective main cells and merged: if the timestamp in the shadow cell is more recent, then its data is copied in the main cell (and the A/W flag is set to

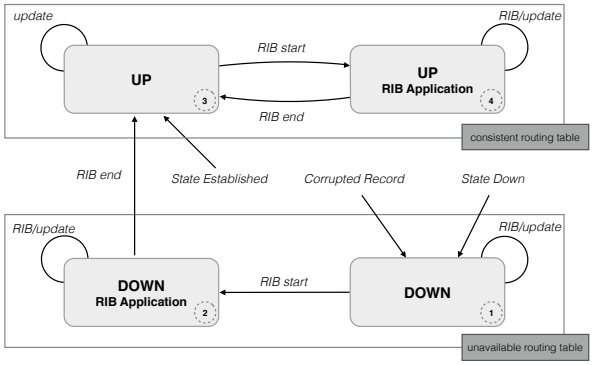

Figure 9: **Finite State Machine for the maintenance of a VP's routing table**. The state chart is made of two macro states that represent the availability or the unavailability of a consistent routing table, each of them having two internal states. Transitions between states are triggered by the reception of a specific BGP record, in italics. The Finite State Machine always starts in the *down* state, then it usually moves to *down-RIB-Application*, and, for the vast majority of time, it oscillates between *up* and *up-RIB-Application*.

"A").

In Figure 9, we describe the process of maintaining a VP routing table as a finite state machine which models the state of the VP. When the plugin starts, the VP's routing table is unavailable (bottom macro-state in Figure 9) and the VP is in state *down* (1). When a new RIB dump starts, the VP's state moves to *down-RIB-application* state (2). During this phase, the plugin populates the shadow cells with the information received from the RIB dump records and the main cells with Update dump records. The VP's state becomes *up* (3) once the entire RIB dump is received, when in this state the routing table is determined to be an accurate representation of the VP's routing table. Each time a new announcement or withdrawal record arrives, it modifies the main cell, whereas if a new RIB dump starts, the VP's state transitions to *up-RIB-application* (4), a state similar to (3) but whereby the RIB dump records modify the shadow information of the cells. Once the RIB ends the shadow and main cells are merged and the VP transitions to state (3) again.

In addition, a corrupted record forces the state to be *down*, as it is not possible to reconstruct a consistent routing table from corrupted data; the reception of an Updates dump record carrying a state message (generated by the collector) with the Established code[4] [36] moves the VP's state to *up*, whereas the reception of a state message carrying any other code notifies that the connection between the VP and the collector is not established, and therefore, the VP is considered *down*[5].

7.1.1 Accuracy and Performance

In order to evaluate how accurate are the VP routing tables that the plugin maintains, we compare the information in the current and shadow cell and we count the number of prefixes that were inactive in the current state and yet are active in the RIB, as well as the number of prefixes that were active in the current state but inactive in the RIB. We find that mismatches are caused by unresponsive VPs for which we do not have state messages (e.g., Route Views), or cases in which the collector does not apply all inbound updates messages before starting its RIB dump, but it applies them afterwards, even if they have been already assigned a timestamp. RIS error probability is 10^{-8}, Route Views error probability is 10^{-5}, where error probability is defined as the number of mismatching prefixes over the sum of all VPs' prefixes across 31 collectors (we computed this probability observing one year of data).

To benchmark the routing-tables plugin, we processed 2 years of data (Aug. 2013 to Aug. 2015) generated by 31 collectors, and find that on average, a day of data is processed in ≈110 minutes (10x faster than realtime). We also performed benchmarking of the system in live-mode (Aug. 19 to Sep. 19 2015): each BGPCorsaro instance requires, on average, 5 minutes to process 5 minutes of data generated by a single collector. Of these 5 minutes, <4 seconds are spent on the actual processing, the remaining time is spent waiting for new data to be available.

7.2 Inter-collector alignment in live mode

At the end of a 1-minute time bin, each BGPCorsaro instance pushes data from its hash table to the BGPViewServer. Such data is merged into a partial BGP view corresponding to its time bin. A BGP view is considered complete when all the BGPCorsaro instances have contributed to it.

We solve the problem of synchronizing data published with variable timing by multiple collectors, in a live monitoring context, by: *(i)* buffering partial BGP views in a sliding window based on their time bins; *(ii)* sliding the window each time data from a new bin arrives; and *(iii)* publishing a BGP view either when all the BGPCorsaro instances have contributed to it (complete view) or when it expires, i.e., its time bin is no longer covered by the window (partial view).

We dimension the length of the sliding window based on empirical observations (over a period of 12 months)

[4]Each collector maintains, for each VP, a Finite State Machine (FSM) that is representative of the status of the BGP session between the VP and the collector itself. When the FSM maintained by the collector for a specific VPs transitions to the Established state, it means the sessions has been established and the VP will start sending BGP update messages shortly.

[5]Route Views does not dump state messages in their Updates dumps, hence it is possible for the plugin to maintain a stale routing table for a VP that is actually down. To mitigate this problem, we also declare a VP down if none of its routes are present in the latest RIB dump from its collector.

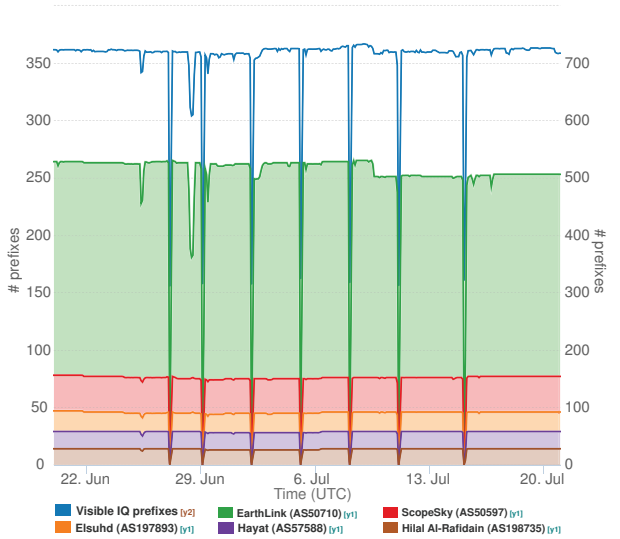

Figure 10: **Visible Iraqi prefixes (June, 20- July, 20 2015).** The blue color indicates the number of prefixes observable in BGP that geolocate in Iraq (y2), the remaining metrics are stacked and show the number of unique prefixes announced by 5 Iraqi providers (y1). There is an observable series of outages that starts on June 27, and ends on July 15: the outages happen at a regular frequency, for a period of about 3 hours, between 2:00am and 5:00am UTC. Such outages have been reported by [5, 15, 18], according to the press the government ordered a complete shutdown of Internet service in the country for three hours.

of the latency at which data providers publish dumps and considering the trade-off with memory footprint: when processing data from all Route Views and RIPE RIS collectors (31), a 30 minute sliding-window buffer requires ≈60GB of memory and causes 99% of BGP views to be published because they are complete rather than expired.

The BGPViewServer is a potential bottleneck in our distributed architecture: as the number of collectors grows, so does the amount of data that the server must receive, process and publish every minute. Although this is not a problem given current data volumes, we architected the server to process each time bin independently of others, allowing multiple server instances to be run (potentially on separate hosts), with BGPCorsaro processes distributing data amongst them in a round-robin fashion.

7.3 BGPViewConsumers

A BGPViewConsumer is an independent process that receives BGP views from the BGPViewServer using a publish-subscribe paradigm. We developed two BGPViewConsumers aimed at near-realtime detection of per-country and per-AS outages (Figure 7). Both consumers select the prefixes observed by full-feed VPs, i.e., those that announce at least 400,000 IPv4 prefixes or 10,000 IPv6 prefixes (similarly to the heuristic in [28]), and continuously monitor their visibility. Specifically, they compute the number of prefixes that are geo-located to each country as well as the number of prefixes announced by each single AS. Each time a BGPViewConsumer finishes processing a BGP view, it sends the results of its computation to a Time Series Monitoring system, which permanently stores them, performs automated detection, and enables data visualization.

In Figure 10, we show the output of the per-country and per-AS outages consumers over a period of 1 month, (June, 20 to July, 20 2015), selecting only the visibility results associated with Iraq and 5 of the biggest Iraqi ISPs. The noticeable drops, in terms of number of visible prefixes, identify a sequence of country-wide Internet outages that the government ordered in conjunction with the ministerial preparatory exams [5, 15, 18].

Similarly, we developed consumers that continuously analyze AS paths in the BGP views, looking for suspicious announcements (e.g., multiple unrelated ASes announcing overlapping portions of the address space, or creating a new edge in the AS graph) as part of a detection system to identify BGP hijacking events [10]. Timely detection of suspicious BGP events enables triggering on-demand data-plane measurements (i.e., traceroutes), which are useful to correlate information from the control and data planes and identify potential mismatches (such as in the presence of man-in-the-middle attacks).

8. CONCLUSIONS

BGPStream targets a broad range of applications and users. We hope that it will enable novel analyses, development of new tools, educational opportunities, as well as feedback and contributions to our platform. We also plan to make available, as Web services, global live monitoring platforms based on the architecture briefly discussed in Section 7.

As mentioned (Section 2), BGPStream development is part of a collaborative effort with other researchers and data providers, such as Route Views and BGPMon, to coordinate progress in this space [7]. We plan to enable new features in the near future (e.g., exposing BGP community attributes) and support for more data formats (e.g., JSON exports from ExaBGP [17]).

9. REFERENCES

[1] Colorado State University. BGPmon. `http://www.bgpmon.io/`, 2015.
[2] S. Anisseh. Internet Topology Characterizationon on AS Level. Master's thesis, KTH, School of Electrical Engineering (EES), Communication Networks, KTH ROYAL INSTITUTE OF TECHNOLOGY, 10 2012.
[3] Apache. Apache Spark. `http://spark.apache.org/`, 2015.
[4] G. D. Battista, M. Rimondini, and G. Sadolfo. Monitoring the status of MPLS VPN and VPLS based on BGP signaling information. In *Network Operations and Management Symposium (NOMS), 2012 IEEE*, pages 237–244. IEEE, 2012.
[5] D. Bernard. Iraqi Internet Experiencing 'Strange' Outages. `http://www.voanews.com/content/`

iraqi-internet-experiencing-strange-outages/2921135.html, 2015.

[6] L. Blunk, M. Karir, and C. Labovitz. Multi-Threaded Routing Toolkit (MRT) Routing Information Export Format. RFC 6396 (Proposed Standard), Oct. 2011.

[7] Claffy, Kc. The 7th Workshop on Active Internet Measurements (AIMS7) Report. *To appear in ACM SIGCOMM Computer Communication Review (CCR)*, 2016.

[8] M. Cosovic, S. Obradovic, and L. Trajkovic. Performance evaluation of BGP anomaly classifiers. In *Digital Information, Networking, and Wireless Communications (DINWC), 2015 Third International Conference on*, pages 115–120. IEEE, 2015.

[9] J. Cowie. The New Threat: Targeted Internet Traffic Misdirection. http://research.dyn.com/2013/11/mitm-internet-hijacking/, 2013.

[10] A. Dainotti. HIJACKS: Detecting and Characterizing Internet Traffic Interception based on BGP Hijacking. http://www.caida.org/funding/hijacks/, 2014. Funding source: NSF CNS-1423659.

[11] A. Dainotti. North Korean Internet outages observed. http://blog.caida.org/best_available_data/2014/12/23/north-korean-internet-outages-observed/, 2014.

[12] A. Dainotti and V. Asturiano. Under the Telescope: Time Warner Cable Internet Outage. http://blog.caida.org/best_available_data/2014/08/29/under-the-telescope-time-warner-cable-internet-outage/, 2014.

[13] A. Dainotti and K. Claffy. Detection and analysis of large-scale Internet infrastructure outages (IODA). http://www.caida.org/funding/ioda/, 2012. Funding source: NSF CNS-1228994.

[14] A. Dainotti, A. King, C. Orsini, and V. Asturiano. BGPStream: a framework for BGP data analysis. https://ripe70.ripe.net/presentations/55-bgpstream.pdf, 2015.

[15] Dyn Research. Iraq has had 12 govt-directed Internet blackouts since 27-Jun. https://twitter.com/DynResearch/status/629393185517666305, 2015.

[16] T. Evens. OpenBMP. http://http://www.openbmp.org/, 2015.

[17] Exa-Networks. ExaBGP. https://github.com/Exa-Networks/exabgp, 2015.

[18] S. Gallagher. Iraqi government shut down Internet to prevent exam cheating? http://arstechnica.com/tech-policy/2015/06/iraqi-government-shut-down-internet-to-prevent-exam-cheating/, 2015.

[19] L. Gao and F. Wang. The extent of as path inflation by routing policies. In *Global Telecommunications Conference, 2002. GLOBECOM'02. IEEE*, volume 3, pages 2180–2184. IEEE, 2002.

[20] V. Giotsas, M. Luckie, B. Huffaker, et al. Inferring complex as relationships. In *Proceedings of the 2014 Conference on Internet Measurement Conference*, pages 23–30. ACM, 2014.

[21] X. Hu and Z. M. Mao. Accurate real-time identification of ip prefix hijacking. In *Security and Privacy, 2007. SP'07. IEEE Symposium on*, pages 3–17. IEEE, 2007.

[22] Q. Jacquemart, G. Urvoy-Keller, and E. Biersack. A longitudinal study of bgp moas prefixes. In *Traffic Monitoring and Analysis*, pages 127–138. Springer, 2014.

[23] E. Karaarslan, A. G. Perez, and C. Siaterlis. Recreating a Large-Scale BGP Incident in a Realistic Environment. In *Information Sciences and Systems 2013*, pages 349–357. Springer, 2013.

[24] D. E. Knuth. *The Art of Computer Programming, Volume 3: (2Nd Ed.) Sorting and Searching*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1998.

[25] C. Labovitz, A. Ahuja, S. Venkatachary, and R. Wattenhofer. The Impact of Internet Policy and Topology on Delayed Routing Convergence. In *Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM), Anchorage, Alaska*, April 2001.

[26] M. Luckie. Spurious routes in public bgp data. *ACM SIGCOMM Computer Communication Review*, 44(3):14–21, 2014.

[27] M. Luckie, B. Huffaker, A. Dhamdhere, V. Giotsas, and k claffy. AS relationships, customer cones, and validation. In *IMC*, Oct. 2013.

[28] A. Lutu, M. Bagnulo, J. Cid-Sueiro, and O. Maennel. Separating wheat from chaff: Winnowing unintended prefixes using machine learning. In *INFOCOM, 2014 Proceedings IEEE*, pages 943–951. IEEE, 2014.

[29] D. Madory. The Vast World of Fraudulent Routing. http://research.dyn.com/2015/01/vast-world-of-fraudulent-routing/, 2015.

[30] R. Mazloum, M.-O. Buob, J. Auge, B. Baynat, D. Rossi, and T. Friedman. Violation of interdomain routing assumptions. In *Passive and Active Measurement*, pages 173–182. Springer, 2014.

[31] NetworkX Developers. NetworkX. https://networkx.github.io, 2015.

[32] U. of Oregon. Route Views Project. http://www.routeviews.org/, 2015.

[33] C. Olschanowsky, M. L. Weikum, J. Smith, C. Papadopoulos, and D. Massey. Delivering diverse BGP data in real-time and through multi-format archiving. In *Technologies for Homeland Security (HST), 2013 IEEE International Conference on*, pages 698–703. IEEE, 2013.

[34] PCH. Packet Clearing House. http://www.pch.net/, 2015.

[35] Quagga. Quagga Routing Software Suite. http://www.nongnu.org/quagga/, 2015.

[36] Y. Rekhter, T. Li, and S. Hares. A Border Gateway Protocol 4 (BGP-4). RFC 4271 (Draft Standard), Jan. 2006. Updated by RFCs 6286, 6608, 6793, 7606, 7607.

[37] P. Richter. Classification of origin AS behavior based on BGP update streams. Master's thesis, Technische Universitat Berlin, 2010. Bachelor Thesis.

[38] RIPE NCC. libBGPdump. https://bitbucket.org/ripencc/bgpdump, 2015.

[39] RIPE NCC. Routing Information Service (RIS). https://www.ripe.net/analyse/internet-measurements/routing-information-service-ris, 2015.

[40] D. Schatzmann, B. Plattner, and W. Mühlbauer. Identification of Connectivity Issues in Large Networks using Data Plane Information.

[41] J. Scudder, R. Fernando, and S. Stuart. BGP Monitoring Protocol. Internet-Draft draft-ietf-grow-bmp-14.txt, IETF Secretariat, Aug. 2015.

[42] N. Spring, R. Mahajan, and T. Anderson. The causes of path inflation. In *Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '03, pages 113–124, New York, NY, USA, 2003. ACM.

[43] C. Q. Sun and P. F. Ding. Optimization Techniques of Traceroute Measurement Based on BGP Routing Table. In *Applied Mechanics and Materials*, volume 303, pages 2062–2067. Trans Tech Publ, 2013.

[44] M. Wählisch, O. Maennel, and T. C. Schmidt. Towards detecting bgp route hijacking using the rpki. *ACM SIGCOMM Computer Communication Review*, 42(4):103–104, 2012.

[45] H. Yan, R. Oliveira, K. Burnett, D. Matthews, L. Zhang, and D. Massey. BGPmon: A real-time, scalable, extensible monitoring system. In *Conference For Homeland Security, 2009. CATCH'09. Cybersecurity Applications & Technology*, pages 212–223. IEEE, 2009.