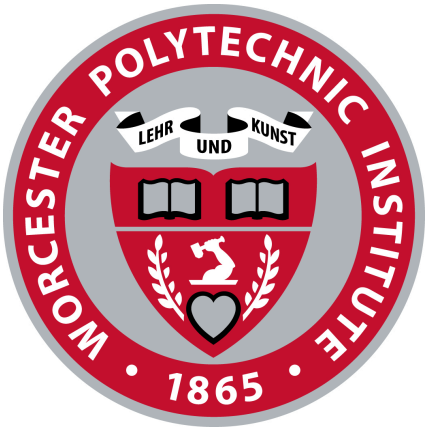
Designing and Developing MIRA:

A Tool for Monitoring Internet Routing Anomalies



A Major Qualifying Project Report
Submitted to the Faculty of
WORCESTER POLYTECHNIC INSTITUTE
In partial fulfillment of the requirements for the
Degree of Bachelor of Science by

Brendan Mannion
Jolene Pern
Taisiia Yakovenko

15 December 2023
Silicon Valley, CA Project Center
Proposal Submitted to:
Professor Mark Claypool

ABSTRACT

Content Delivery Networks (CDNs), like Fastly, distribute traffic across the Internet for better reliability and load distribution. CDNs need to identify significant changes in external routing posture to better balance their traffic. We designed and developed MIRA (Monitoring Internet Routing Anomalies) to analyze public routing data and identify when large influxes of Border Gateway Protocol (BGP) updates occur during a short period of time, indicating a significant route change. MIRA provides a comprehensive view on routing patterns for each user-specified subscription by graphing the number of incoming messages per minute, determining anomalies, and displaying every message associated with the anomalies. MIRA also keeps historical data to better identify anomaly patterns.

ACKNOWLEDGMENTS

Our team extends heartfelt gratitude to Fastly, our generous sponsor, for affording us this incredible opportunity to engage, learn, and collaborate within the dynamic environment of Fastly. A special appreciation goes to our exceptional Fastly mentors, Stephen Strowes, Salman Saghafi, and Ashwin Pai, for their unwavering guidance and support throughout the project. We would also like to thank the entire Fastly team for their warm welcome and valuable contributions that significantly contributed to the success of this project. Lastly, immense appreciation goes to our MQP advisor, Professor Mark Claypool, for orchestrating this project and providing invaluable feedback and guidance every step of the way.

Table of Content

LIST OF FIGURES

1.0 INTRODUCTION

Routing is an important part of a functioning Internet. Internet routing is the dynamic process by which routes are shared across tens of thousands of networks, enabling communication between a vast array of interconnected services and devices. Content Distribution Networks (CDNs), such as Fastly, operate in complex infrastructure in multiple physical locations, and then distribute traffic across those locations to increase reliability and load distribution. CDNs rely on Internet routing to balance their traffic, and so identifying significant changes in the routing topology can be important to their operations.

While Fastly has internal tools to track and record its intended routing posture from the inside, external data sources are critical for understanding how other networks access Fastly's network from the Internet. One type of event that can indicate significant change in routing is the rapid occurrence of Border Gateway Protocol (BGP) updates during a short period of time. Currently, no public tools attempt to detect these influxes of BGP updates. The timely identification of such BGP update influxes can be helpful for stabilizing the performance of Fastly's network, as it enables proactive response to potential disruptions.

The goal of our project is to build a tool that detects, analyzes, and displays changes in Internet routes to track change patterns. The project created MIRA (Monitoring Internet Routing Anomalies), a tool to perform the analysis and a visual interface to make the results easier to interpret. The aim of the tool is to process live and static public BGP data and detect when large amounts of BGP updates are happening during a short period of time. Another goal of the project is to create visualizations of the BGP updates over time for specific user-defined subscriptions, to make the results of our tool more digestible.

Our project looks at the evaluation of our tool by examining the performance of our anomaly detection algorithms. This involves making use of the static data portion of our tool to replay historical events or known BGP anomalies and evaluate whether or not our tool detects similar outliers. We also look at linear regression and other statistical models to fine tune the optimization parameters of one of our anomaly detection algorithms to get more accurate results based on the specific BGP data.

This report describes the research, methodology, implementation, evaluation, future work, and concluding remarks for our tool MIRA. Chapter 2 provides the necessary background for understanding Internet routing in the context of our tool. Chapter 3 discusses our team's approach in project management to develop this tool. Chapter 4 goes through the full design and implementation of MIRA, examining both the back-end and front-end of our tool. Chapter 5 examines the performance of our anomaly detection algorithms on BGP data. Chapter 6 looks at future work to improve upon the tool we created. Finally, Chapter 7 summarizes the conclusion for our project.

2.0 BACKGROUND

This section aims to provide a basic understanding of computer networking within the context of our tool.

2.1 Networks

In the context of routing, networks are groups of interconnected devices and routers that are organized together to form a single entity with a common routing policy. The Internet is a network of networks. Internet routing occurs between these networks, directing data to pass from network to network all across the Internet, visualized in Figure 1.

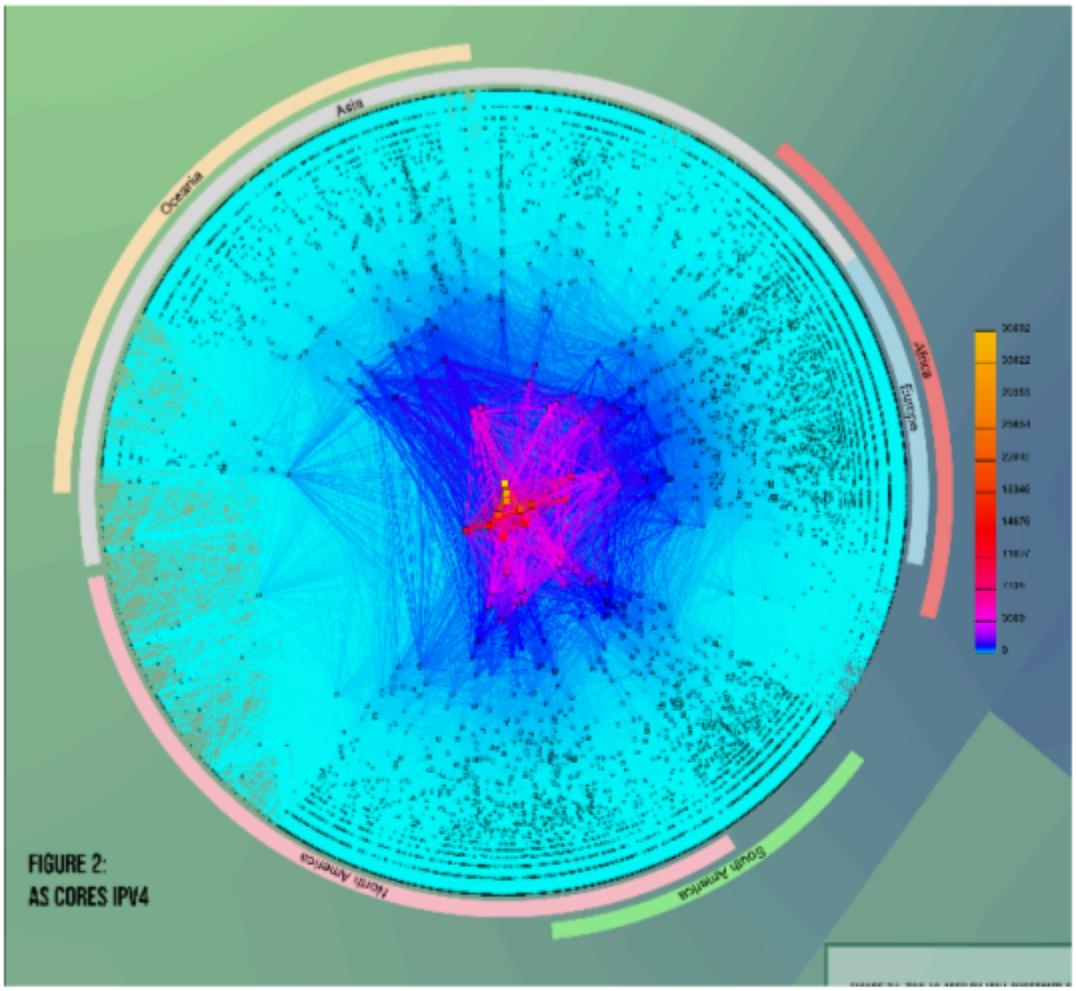


Figure 1: Visual representation of the Internet as a network

Networks can be represented by graphs with nodes and links that serve as paths between nodes. Routing is the process by which routers inform each other of valid paths to various destinations, each building up a local routing table that it uses to determine the best paths available [1]. Since links between nodes can be of various lengths and there are several ways nodes link to one another, there are numerous routing algorithms to determine the shortest or most efficient path to transfer the data from one node to another.

In inter-domain routing, networks are referred to as Autonomous Systems (ASes); each Autonomous System is represented in the routing system by an Autonomous System Number

(ASN). Within a network, specific details are shared to local network operations. However, on a larger scale, these details are typically neither shared nor necessary to other networks. This abstraction allows individual organizations to have control over their section of the Internet. These organizations can also make key decisions regarding routing policies, traffic management, and network administration for their collection of IP addresses. Networks are often dynamic, due to new networks and paths continuously being opened and closed.

There are several ways in which networks can connect to each other. The most common is through an Internet Service Provider (ISP). An Internet Service Provider (ISP) is a company or organization that provides access to the Internet for individuals, businesses, and other entities. ISPs serve as intermediaries that connect their customers to the global Internet infrastructure, which enables users to access online content, send and receive emails, browse websites, and engage in various online activities. A network can have either one ISP or multiple. While there are more benefits to working with multiple upstream providers, it is often cheaper and simpler to opt into only using one. An increased number of upstream providers allows for more routing options which can be more efficient.

In some special cases, two networks may be under the same ISP with their ISP being the only link - these networks are considered peers. If these two networks send an abundance of traffic to each other, there is often a common exchange point (IXP) that physically connects two networks together to easily send information back and forth. This IXP eliminates the need to go to and from the ISP, saving time and cost. This is called peering.

2.2 Border Gateway Protocol (BGP)

Border Gateway Protocol (BGP) operates at the level of autonomous systems and is used to efficiently route traffic through the network. In order to join BGP, a network must have an IP address space to route, and an ASN to send it from [2]. IP address spaces and ASNs are both obtained from one of the five Routing Internet Registries (RIRs, Afrinic, etc…). BGP uses a path vector table which maintains all the attributes of various routes and policies associated with specific regions which are defined by system administrators as shown in Figure 2. Using these tables, BGP uses routing algorithms to route efficiently given the configured policies.

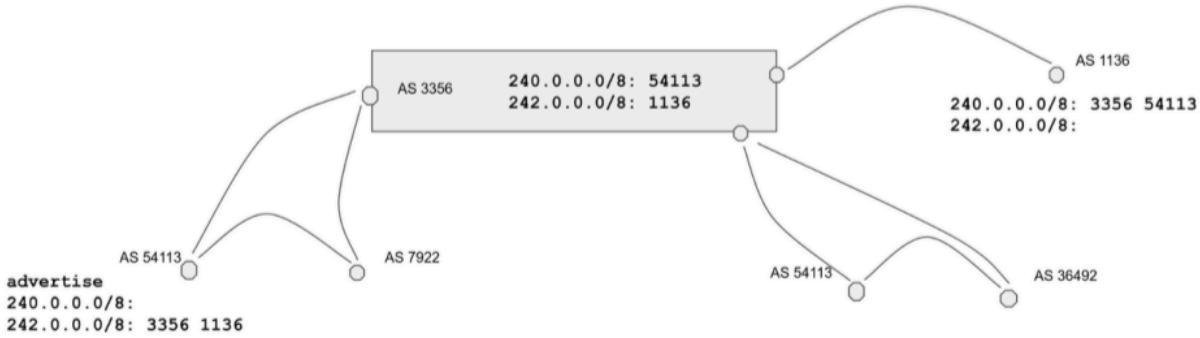


Figure 2: Autonomous Systems and their relationships with one another

There is always information exchange between routers via peering sessions. Such sessions are used to ensure the most up-to-date state of each route, potential path, and policy updates for each. BGP allows sharing of all information between large Internet Service Providers (ISPs), creating a well-connected and efficient global net. ISPs may configure permissions for the ASNs to serve various business purposes, often to minimize routing costs. More specifically, transit networks can be used by peer networks to avoid using costly upstream networks which are necessary to connect to the worldwide web. Peer networks set up symbiotic relationships

allowing them to share traffic directly rather than via another provider, leading to reduced (or, often, near-zero) costs. The information stored in the routing table as well as information about all the entities in the global network is used with the number of messages to further analyze for purposes of this project.

2.3 Public Data Sources

There are two large route collector projects active on the Internet today, RouteViews [3] and the Réseaux IP Européens Routing Information Service (RIPE RIS) [4]. Both of these projects aim to obtain and share information about the global Internet routing system. To do this, they operate a network of globally distributed route collectors, strategically positioned at various vantage points across the Internet infrastructure that other operational networks peer with to share routing data. These route collectors gather an extensive dataset of BGP routing information from numerous sources, including both transit providers and peers, demonstrated in Figure 3. These datasets serve as a valuable resource for our project, offering an extensive view of how routes are advertised and withdrawn across the Internet.

Figure 3: RouteViews collector map

Routeviews, shown in Figure 3, is a project funded by the University of Oregon's Advanced Network Technology Center as shown in Figure 3. RouteViews offers public static Updates files, offering a complete historical archive of routing updates from each of their collectors [3]. We chose to use the RouteViews Update files as our standard input for static data in our tool.

Another source of data for our project is RIPE RIS, a not-for-profit organization that provides several services to support the infrastructure of the Internet [4]. Accessing data from RIPE RIS requires establishing connections to their route collectors and utilizing their publicly available data feeds. RIS Live is a stream of real-time BGP messages encoded as JSON data structures and can be easily accessible through a websocket interface. Due to its simple websocket connection, we chose to use RIS Live as our standard input for live data in our tool.

3.0 METHODOLOGY

This section examines the strategic methods we used throughout our project to optimize efficiency.

3.1 Agile Methodology

Agile methodology is a common approach to software development and project management utilizing iterative cycles to develop features in a flexible and collaborative way. This methodology uses sprints, which are a predetermined time span, where a set of tasks are sized and planned to be completed in that time with a deliverable working product at the end of each sprint. This method fosters incremental delivery of smaller but functional features that lead to a successful final product. The team opted for week-long sprints to allow for more review and iterations. Each sprint consists of a sprint planning session, daily standups, and a retrospective session.

The team used Trello, an online project management tool, to keep track of tasks and their status throughout sprints. There were four statuses that a task had: Backlog, Sprint Backlog, Development, and Complete. The Backlog consists of tasks that have not been implemented and not in the current sprint. The Sprint Backlog consists of tasks that have been planned to be completed in the current sprint, but have not been started. The Development consisted of tasks that are in progress in the current sprint. Lastly, the Complete column consisted of finished tasks from the current sprint.

3.2 Sprint Planning

The sprint planning sessions took place at the beginning of each week, since the team chose week-long sprints. The meeting served to define tasks in the Sprint Backlog as well as their corresponding story points to complete in the new sprint. Typically, each Trello card represents a user story or feature as shown in Figure 4. To choose tasks for the current sprint, the team could either pull tasks from the Backlog or write new tasks and directly put them in the Sprint Backlog.

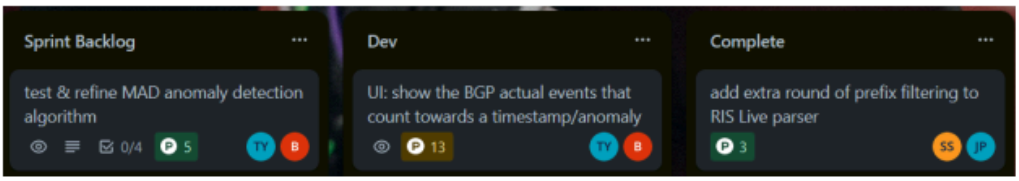


Figure 4: Screenshot of Trello board

Story points were an estimate for the size of a particular task. Story point values were Fibonacci numbers, as to express how small or large the task is in relation to other tasks. After the first sprint, the team was able to use the total number of completed story points from previous sprints to estimate how much work can be done in the new sprint. Sprint planning allowed the team to view the project holistically and decide which features can be implemented next.

3.3 Daily Standups

The daily standups were a 15 minute meeting at the beginning of each work day. The meetings were used as a method to keep everyone on the team updated about progress and potential roadblocks. During these meetings, any of the roadblocks were addressed by scheduling further meetings or having a brief discussion on the spot. Such an approach allowed us to keep the work progressing at the best pace possible.

3.4 Retrospectives

The retrospective occurred weekly on Fridays at the end of the day. The meeting served as a way to reflect on the work week, address all the tasks that were completed, address any concern related to the project, and demonstrate the progress.

The meeting started by putting down all the positive, neutral, and negative aspects of work for the week. Then, all the team members went through all the points and anonymously voted for the ones that they agreed with. Then, the points that got the most votes were discussed together as a team. The discussion addressed the concerns and commented on all the other points which would have resulted in creating action items for the following weeks.

The demonstrations assured that at the end of each week there was a working portion of the code that ran some of the project specifications. The demonstrations were also accompanied by a presentation describing the details of implementation and any changes from previous weeks.

4.0 IMPLEMENTATION

This section discusses the implementation of our tool, Monitoring Internet Routing Anomalies (MIRA). This includes both the back-end and front-end parts of our system, and describes the design. Figure 5 illustrates the system architecture and the primary steps of our tool.

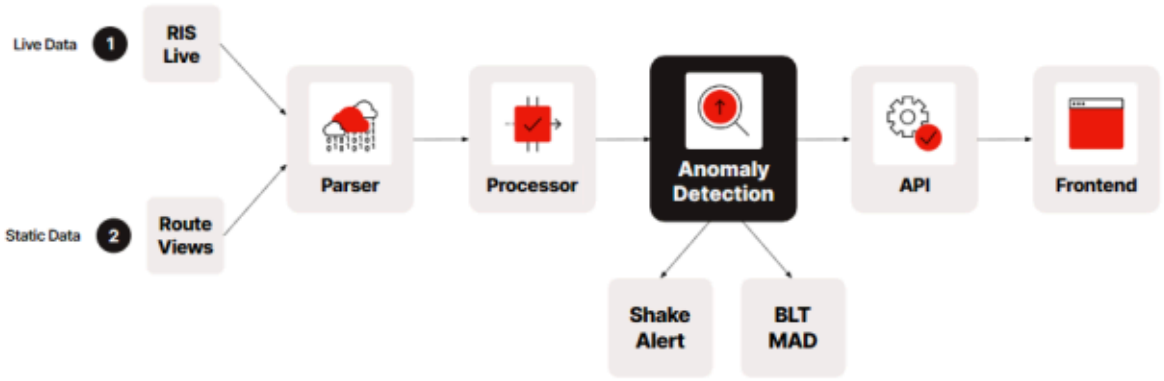


Figure 5: Overview of system architecture

4.1 Back-end

For our back-end system, we used the programming language Golang (Go). We broke the back-end down into different services each with a specific purpose. These services include: configuration, parsing, message processing, analysis, and API.

4.1.1 Configuration

Users are able to provide a Configuration JSON file to tailor MIRA to their specific needs. As demonstrated in Figure 6, users can provide the path to a configuration JSON file via a

command line flag. If no configuration is provided, the tool uses a default configuration file and detects anomalies in live BGP data linked to Fastly's ASN.

```
go run main.go -config="path_to_config_json"
```

Figure 6: Passing a JSON configuration file via command line

We defined a Configuration structure to encapsulate all of the parameters that are configurable, shown in Figure 7. Users must specify the analysis of either live or static data by providing a "dataOption " in the configuration JSON file. If static analysis is chosen, the path to a static file is also required for MIRA to run on that file. If live analysis is chosen, at least one subscription must be provided in an array of a Subscription structure, which is to be applied to the RIS Live service and filter BGP messages. Users can also specify which anomaly detection algorithm is performed, which is set by default to perform both ShakeAlert and BLT MAD. The ShakeAlertParam, MaxBuckets, and WindowSize are all optional parameters that can be set, but are set to a default value if omitted from the configuration file. Lastly, a static file link can be given to load static files from a RouteViews link.

```
type Configuration struct {
    FileInputOption string            `json:"dataOption"`
    StaticFile      string            `json:"staticFilePath"`
    Subscriptions   []SubscriptionMsg `json:"subscriptions"`
    Algorithm       string            `json:"anomalyDetectionAlgo"`
    ShakeAlertParam int               `json:"shakeAlertParameters"`
    MaxBuckets      int               `json:"maxBuckets"`
    WindowSize      int               `json:"windowSize"`
    URLStaticData   string            `json:"staticFilesLink"`
}
```

Figure 7: Configuration parameters stored in a structure

Once a configuration JSON file is loaded and placed into our Configuration structure, MIRA performs validation to ensure all inputs are valid and all required parameters have been given. If there are missing or invalid parameters, an error may be thrown or the parameter is set to a default value depending on the parameter requirements. Documentation is supplied in the source code to aid users in learning which parameters are required and valid input.

4.1.2 Parsing

Our BGP Message Parser is designed to retrieve both live and static update feeds from public BGP routing data sources. Since static and live data are disparate formats, MIRA normalizes them into a BGP Message structure, extracting the vital attributes of each message that are necessary for the rest of the tool to detect BGP anomalies. The final goal of our BGP Parser is to send these parsed BGP Messages to a channel, which is used to retrieve the messages later in the program.

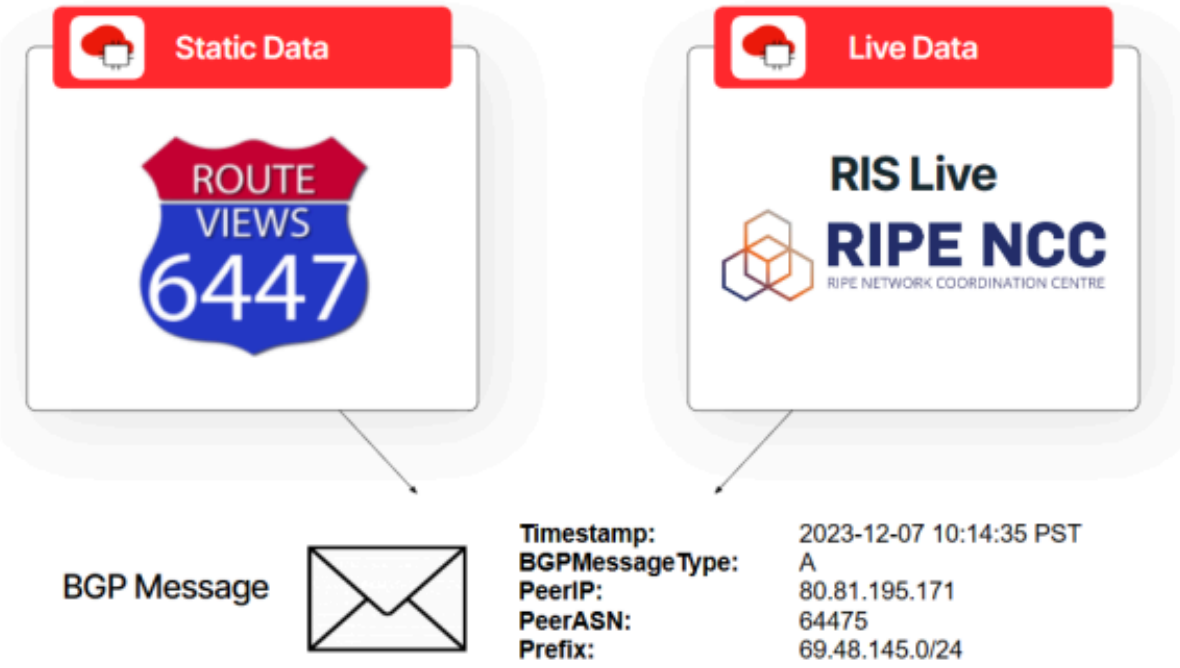


Figure 8: BGP Messages are parsed from both static and live update feeds

MIRA parses both live data and static data from public data sources. Live data provides real-time BGP updates that can be used for detecting events as they happen. On the other hand, extracting data from static files allows users to look back into historical archives and analyze events that have previously happened. Additionally, static file parsing is helpful for testing our tool's anomaly detection algorithms. By running our tool on static data with known events, we were able to check the accuracy of various anomaly detection algorithms on a range of parameters and optimize parameter selection.

```
type BGPMessage struct {
    Timestamp       time.Time
    BGPMessageType  string
    PeerIP          netip.Addr
    PeerASN         uint32
    Prefix          netip.Prefix
}
```

Figure 9: BGP Message structure

4.1.2.2 Static Data

RouteViews provides static Update data streams in the form of bz2 files. MIRA uses bgpdump [5], a tool that converts BGP routing data to a string, to read all of the updates in a given static file. MIRA then parses each message to match the BGP Message structure we defined and passes the parsed messages into a channel to be called by the Message Processor next. Figure 10 shows one example of a withdrawal message from bgpdump, which is then parsed to match the BGP Message structure.

```
BGP4MP_ET|1638317694.706880|W|2001:504:36::6:1481:0:1|398465|2a10:cc42:131c::/48
```

Figure 10: Example of withdrawal update message from bgpdump before parsing

4.1.2.2 Live Data

RIS Live provides a websocket interface to connect to a stream of BGP updates from all of their route collectors in real-time [6]. Figure 11 shows an example of a BGP update message that is received from the websocket. These messages are retrieved as JSON messages, thus

MIRA normalizes the JSON message to match the BGP Message structure.

```
{
    "type":"ris_message",
    "data": {
        "timestamp":1695269583.730,
        "peer":"217.29.66.158",
        "peer_asn":"24482",
        "id":"217.29.66.158-018ab5f0fb720005",
        "host":"rrc10.ripe.net",
        "type":"UPDATE",
        "path":[24482,6939,38040,23969],
        "community":[[24482,2],[24482,200],[24482,12000],[24482,12040],[24482,12041],[24482,20100]],
        "origin":"IGP",
        "med":0,
        "announcements":[{"next_hop":"217.29.66.158","prefixes":["1.1.249.0/24"]}],
        "withdrawals":[]
    }
}
```


Figure 11: Example of RIS Live message before parsing

One useful feature of RIS Live is the feature of subscriptions, in which connections can subscribe to listen to certain types of BGP Messages in specific spaces of the Internet. The primary use case of MIRA is to monitor a certain address space, such as the Fastly ASN 54113. Thus, we defined a SubscriptionMsg structure, as shown in Figure 12, that encapsulates all aspects of a subscription that a typical user of MIRA would care about subscribing to: host, peer, ASN, and prefix. MIRA allows users to monitor multiple subscriptions simultaneously, allowing users to compare various address spaces and anomalies in these spaces. To do this, MIRA utilizes multi-threading processes to manage multiple connections to the RIS Live websocket.

```
type SubscriptionMsg struct {
    Host   string `json:"host,omitempty"`
    Peer   string `json:"peer,omitempty"`
    Asn    int    `json:"asn,omitempty"`
    Prefix string `json:"prefix,omitempty"`
}
```

Figure 12: Subscription Message structure

4.1.3 Message Processing

The message processing step prepares data and initiates analysis once there is a specified amount of data. The system also needs to efficiently store messages and organize them in a way that makes it easier to pass into our following steps. A final goal is to ensure that old data gets evicted that is no longer being used to avoid potential memory issues from running over long periods of time.

As mentioned in the previous section, our message processor receives messages from our parsing system using a channel in Go. We needed a structure that would hold onto these messages and associate them with a specific timestamp for further use in our analysis algorithms. This structure also needed to differentiate which subscription the messages are from.

```
type Window struct {
    Filter    string
    BucketMap map[time.Time][]BGPMessage
}
```

Figure 13: Window structure

As depicted in Figure 13, we defined a Window structure to store all the different messages that came from a specific filter or subscription. This Window structure also has a BucketMap which is a map of a timestamp to an array of BGPMessages which allows us to continuously store all the messages for a specific timestamp, allowing us to keep a history of potential anomalous messages as well as creating an easy to use structure to pass into our analysis.

```
// Stores windows that will be used for analysis
windows := make(map[string]*common.Window)

// Read messages from channel
for msg := range msgChannel {
    // Get current window from map if it exists
    currWindow, exists := windows[msg.Filter]

    // If the filter is not already a window, create a new window for it
    if !exists {
        window := common.Window{
            Filter:    msg.Filter,
            BucketMap: make(map[time.Time][]common.BGPMessage),
        }

        // Add window to the map
        windows[msg.Filter] = &window

        // Update currWindow to point to the newly added window
        currWindow = &window
    }
```

Figure 14: Selecting the correct window when processing messages

Figure 14 shows a closer look at how MIRA makes use of these Window structures. The tool first initializes a map of a filter string to a Window object. Whenever MIRA reads in a

message it first looks at the filter string and checks if there already exists a Window for this filter, if not the system creates a new Window and continues processing.

```
// Round timestamp to nearest minute
messageBucket := msg.BGPMessage.Timestamp.Truncate(60 * time.Second)
```

Figure 15: Rounding timestamp to minute

Each time MIRA reads a message, it looks at the message's timestamp and rounds it to the nearest minute, as seen in Figure 15, to keep track of all the different BGP messages that occurred in that one minute.

Now the tool can query our BucketMap structure and see if there are any messages already stored for this minute. If there are not any messages stored already, then MIRA creates a new entry in the map with this timestamp mapped to an array of the new message to be added. If the timestamp is already in the map, we simply append this new message to the array of messages.

Another main design point of our processor system was deciding when to call analysis. With a program with only live data, it may make sense to have a timer run and once each minute goes by, check if there is enough data, and then perform analysis. However, due to our use of channels and a common message structure, our program can handle live and historical data at once, so this timer strategy would not suffice. With the use of historical data, our program would receive hours or even days worth of messages all arriving at once, so we needed to ensure the tool could handle that.

We proceeded with a design in which, upon receiving a message with a new timestamp (no existing entry in the map), the tool checks if our structure is ready for analysis. This check involves comparing the number of elements stored in the map with the configuration parameter for window size. If MIRA has at least a window size number of elements in the map, the tool proceeds to perform analysis using the specified window.

```
// If we at least maximumBuckets in bucketMap we can perform analysis
if len(currWindow.BucketMap) >= maximumBuckets {

    // First want to remove timestamps out of scope so len(bucketMap) == maximumBuckets
    minimumTimestamp := messageBucket.Add(-maximumTimespan)
    for timestamp := range currWindow.BucketMap {
        if timestamp.Before(minimumTimestamp) {
            fmt.Println("Expired bucket: ", timestamp)
            delete(currWindow.BucketMap, timestamp)
        }
    }

    // Now window is ready for analysis
    analyze.AnalyzeBGPMessages(*currWindow, config)
}
```

Figure 16: Evicting old data and calling analysis

Another key part of this step involves the cleanup of old data in our structure, demonstrated in Figure 16. In the case where the number of elements in the map is greater than the our configuration parameter for window size, we need to remove elements that are out of scope. We accomplish this by calculating what the smallest timestamp should be in the map based on the newest timestamp we just received. Then we go through each element in the map removing elements that are smaller than this timestamp. This ensures that our program will not run into memory issues by storing too much data about a specific subscription, and also enforces that our analysis algorithm always receives windows of the same size.

4.1.4 Analysis

The goal of our Analysis system is to take the information from our message processor and run two analysis algorithms: Shake Alert and BLT MAD. After receiving results from the analysis algorithms, the system logs outputs to the console and stores them for future use.

Both of the analysis algorithms run on an array frequency counts that represent the number of announcement or withdrawal messages that occurred during each minute of the given Window structure. With the use of the Window object coming from the message processor, each entry of the BucketMap is iterated through which creates an array of frequencies by taking the length of each message array in the map.

```
// Turn map into sorted array of frequencies by timestamp
sortedFrequencies, sortedTimestamps := GetSortedFrequencies(lengthMap)

// Call both analysis algorithms
bltOutliers, bltIndexes := blt_mad.BltMad(sortedFrequencies, OPTPARAM)
shakeAlertOutliers, shakeAlertIndexes := shake_alert.FindOutliers(sortedFrequencies)
```

Figure 17: Both of the analysis functions are called using the arrays of frequencies and given sensitivity parameters

4.1.4.1 ShakeAlert

ShakeAlert is an outlier detection algorithm developed by Edgio [7]. The algorithm runs on an array of frequencies of BGP messages in each minute and the similarity to determine if a particular point is an outlier. The algorithm calculates the radius for a given data set by finding the difference between the 95th and 5th percentiles. Then, for each data point the number of data points within the radius to the given one is found resulting in the number of neighbors for that

particular point. If the number of neighbors for that point is less than than the minimum required number of similar points given earlier, then the point is an outlier. Figure 18 below uses an example array [1,1,1,1,900,1,1000,1,1] to follow the calculation that ShakeAlert will perform to calculate the outliers.

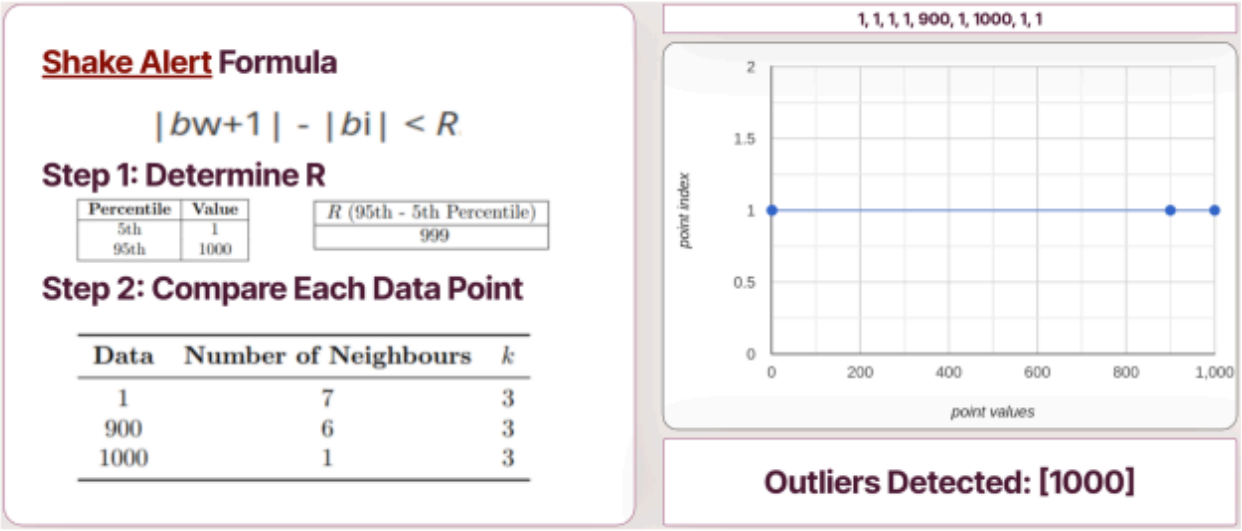


Figure 18: An example of calculations that ShakeAlert performs to determine outliers.

The radius is found by calculating the difference between the 95th and the 5th percentile to use it as means of comparison as described in Figure 19.

```
percentile_5 := findPercentile(data, p: 5.0)
percentile_95 := findPercentile(data, p: 95.0)

R := percentile_95 - percentile_5
```

Figure 19: Use the 95th and 5th percentile calculations for radius

After completing the preliminary calculations described above, the outliers are found by iterating through every point in the data and calculating the number of neighbors which is then compared to the minimum required number of neighbors as described in Figure 20.

```
//iterate through each time bin
for i := 0; i < len(data); i++ {
    currentBin := data[i]

    // count the number of neighbors within radius R
    R := findR(dataCopy)
    neighborCount := 0
    for _, bin := range data {
        if math.Abs(float64(bin-currentBin)) < R {
            neighborCount++
        }
    }

    // if there are fewer than k neighbors, the current bin is an outlier
    if neighborCount < k {
        outliers = append(outliers, currentBin)
        indexes = append(indexes, i)
    }
}
```

Figure 20: Iterate through all the data points in a given set and determine which of the points are outliers based on comparisons using ShakeAlert methodology

4.1.4.2 BLT MAD

Our implementation for the BLT MAD outlier detection algorithm was a simplification of an algorithm proposed by the BGP Update Labeling tool [8]. The algorithm proposed relies on calculating the median and the mean average distance of the data set to compare each individual point to determine outliers. BLT MAD also uses a configurable parameter to determine the

weight of mean average distance value in determining the outliers. The algorithm follows the steps described in Figure 21.

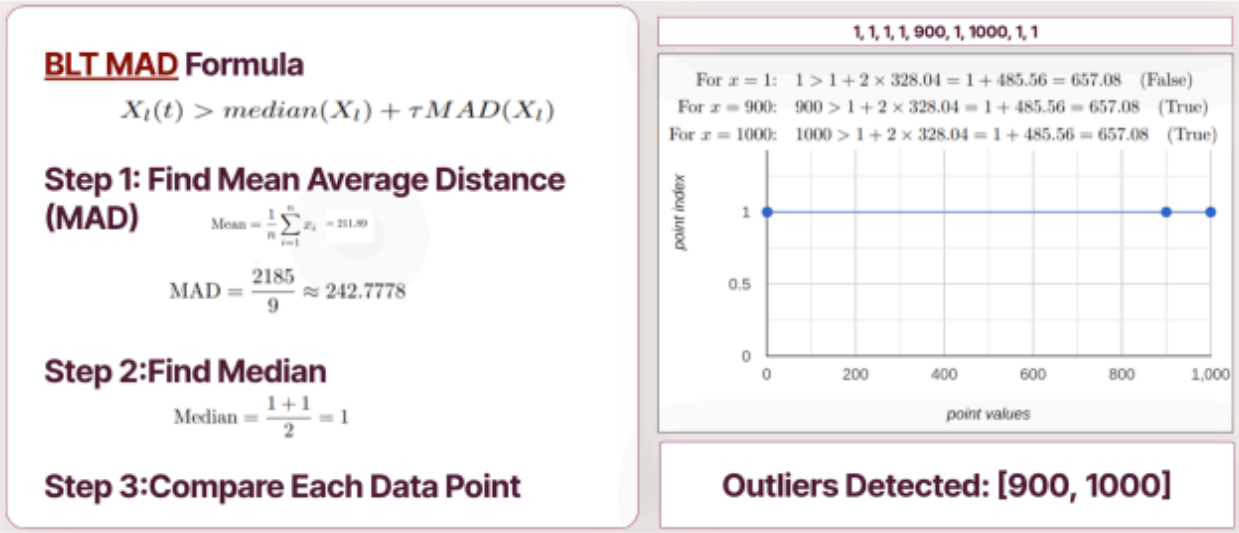


Figure 21: An example of calculations that BLT MAD performs to determine outliers.

Our implementation performs calculations to find the mean and the mean average distance of the data set. Those values in combinations with the sensitivity parameter are then used to determine whether or not each individual point is an outlier.

```
//calculate blt formula
//everything is based on the noZeroData since we are looking at spikes rather than lack of messages
med := FindMedian(noZeroData)
m := Mad(noZeroData)

bltScore := math.Abs(med - tau*m)
for i, value := range noZeroData {
    if value > bltScore {
        outliers = append(outliers, value)
        indexes = append(indexes, i)
    }
}
```

Figure 22: Iterate through each point in the given dataset to determine the outliers

4.1.5 API

The main purpose of our API is to allow users to get a structured form of the data that our program collects and analyzes. Another goal is to enable users to query specific information.

```
// Set up http server for API
fs := http.FileServer(http.Dir("static"))

// Set up different endpoints
http.Handle("/", fs)
http.HandleFunc("/data", getData)
http.HandleFunc("/subscriptions", getSubscriptions)
http.HandleFunc("/frequencies", getFrequenciesFromSubscription)
http.HandleFunc("/outliers", getOutliersFromSubscription)

log.Println("Server started on port 8080")
err = http.ListenAndServe(":8080", nil)
if err != nil {
    log.Fatal("Server error:", err)
}
```

Figure 23: Setting up http server for API

In order to implement our API using Go, we used the net/http library [9]. This library enables http server on localhost in order to receive requests to our API. We set up our API on port 8080, so that users running the program can access our APIs from localhost:8080. The tool currently offers five different endpoints where users can get various information about our program. Figure 23 shows an overview of how we made use of the net/http library to set up our different endpoints and start listening on port 8080.

The root (/) endpoint of our program is used to serve the static files that our program provides. This currently includes the code for our front-end, more specifically the index.html and script.js file. This API allows users who visit localhost:8080/ after starting up the program to see the front-end visualization of our tool.

The /data endpoint provides a structured format of all of the data that our program collects about the various subscriptions that they are listening to. This request gets fulfilled as shown in Figure 24, by our program going to the results that we store in our analysis system, and getting the map of all different results for each subscription and encoding this into JSON to return to the user.

```go
func getData(w http.ResponseWriter, r *http.Request) {
    fmt.Println("Data request")
    w.Header().Set("Content-Type", "application/json")
    err := json.NewEncoder(w).Encode(analyze.ResultMap)
    if err != nil {
        http.Error(w, "Error encoding JSON", http.StatusInternalServerError)
        return
    }
}
```

Figure 24: Get /data method implementation

```
{
    "{\"path\":\"54113$\"}": {
        "Filter": "{\"path\":\"54113$\"}",
        "AllOutliers": {
            "2023-12-14T18:47:00Z": {
                "Timestamp": "2023-12-14T18:47:00Z",
                "Algorithm": 3,
                "Count": 169,
                "Msgs": [···
                ]
            },
            "2023-12-14T18:48:00Z": {···
            },
            "2023-12-14T18:49:00Z": {···
            }
        },
        "AllFreq": {
            "2023-12-14T18:47:00Z": 169,
            "2023-12-14T18:48:00Z": 230,
            "2023-12-14T18:49:00Z": 366
        }
    },
    "{\"prefix\":\"151.101.0.0/16\"}": {
        "Filter": "{\"prefix\":\"151.101.0.0/16\"}",
        "AllOutliers": {
            "2023-12-14T18:47:00Z": {
```


Figure 25: /data example response

As seen in Figure 25, the /data endpoint gives all the information about each different subscription the tool is listening to, including the different outliers detected for that subscription, as well as the frequency counts for the number of messages in each minute timestamp for each subscription.

The /subscriptions endpoint of our program is used to get the string format of the different subscriptions that the program is listening to. These can be helpful for querying specific information about various subscriptions and can remind users during the running of the application what different filters they have selected. Similar to the /data endpoint, this endpoint

also makes use of the result map we store in analysis and gets the filter string from each of the entries in the map, and returns them as a JSON encoded array of strings. An example set of these strings are depicted in Figure 26.

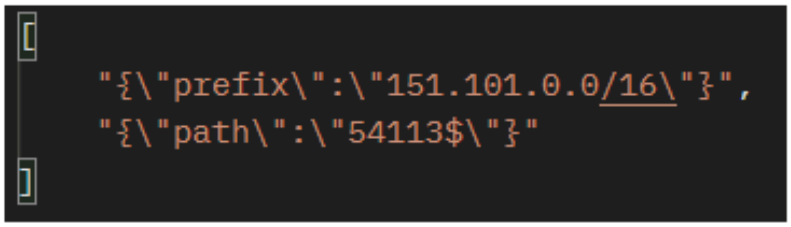


Figure 26: /subscriptions example response

The /frequencies endpoint gets the frequency information about a specific subscription. This endpoint requires an additional query parameter called subscription which is one of the strings provided in the above /subscriptions example. Figure 26 shows an example of a call to /frequencies endpoint along with an example query parameter. When calling this function with a proper subscription string, the user receives a list of the different minute timestamps that have been collected for that subscription, along with the accompanied number of messages that were observed during each minute timestamp. An example of these message counts can be seen in Figure 28 from the example request called in Figure 27.

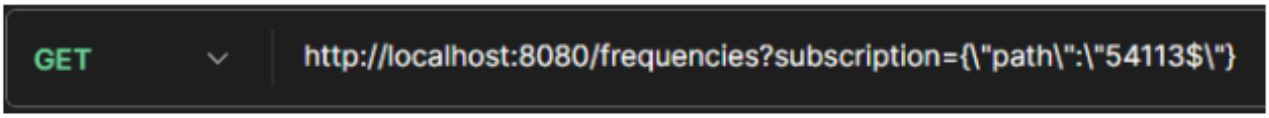


Figure 27: /frequencies example query

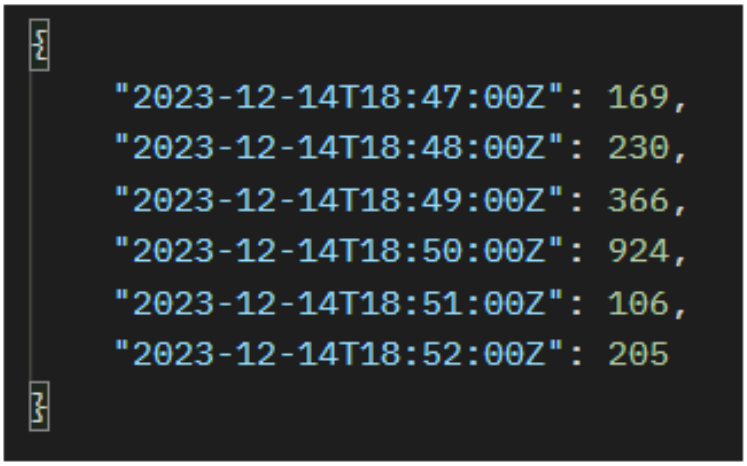


Figure 28: /frequencies example response

Similar to the frequencies endpoint, the /outliers endpoint also takes in the same subscription query parameter and gives specific information about all the different outliers collected for that specific subscription. Figure 29 showcases an example of the outliers endpoint with one of the previously mentioned subscription strings from Figure 26. The response gives a list of all the different timestamps of outliers collected, along with more specific information such as which algorithms detected it (1 = BLT MAD, 2 = Shake Alert, 3 = Both Algorithms), the number of messages that occurred during that minute, and finally a list of all the different BGPMessages that were apart of that anomaly. Figure 30, gives an example of the API response from querying the URL in Figure 29.

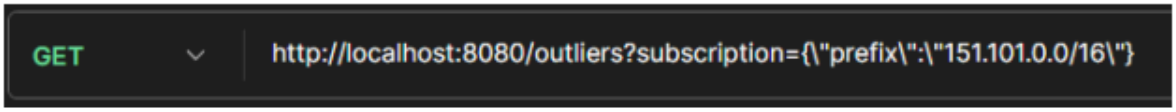


Figure 29: /outliers example query

```
{
    "2023-12-14T18:47:00Z": {
        "Timestamp": "2023-12-14T18:47:00Z",
        "Algorithm": 2,
        "Count": 7,
        "Msgs": [...
        ]
    },
    "2023-12-14T18:48:00Z": {
        "Timestamp": "2023-12-14T18:48:00Z",
        "Algorithm": 2,
        "Count": 58,
        "Msgs": [
            {
                "Timestamp": "2023-12-14T18:48:15.380000114Z",
                "BGPMessageType": "A",
                "PeerIP": "190.112.55.254",
                "PeerASN": 264845,
                "Prefix": "151.101.176.0/22"
            },
            {
                "Timestamp": "2023-12-14T18:48:21.400000095Z",
                "BGPMessageType": "A",
                "PeerIP": "186.211.136.32",
                "PeerASN": 14840,
                "Prefix": "43.249.74.0/24"
            },
```


Figure 30: /outliers example response

4.2 Front-end

Our front-end runs on a local server on port 8080 and provides a UI for the user to select the displayed graph and outlier messages for each of the subscriptions for which the data is currently being collected from. The user is able to select the subscription of interest with a drop down menu which pops up a new tab in the browser showing all the data for that subscription.

Figure 31:Drop down menu allows the user to select the filter they want to monitor

The drop down is populated as the data for each of the filters is collected and stored in the /data endpoint every three seconds as described in Figure 32.

```
const url :string  = 'http://localhost:8080/data';
fetchByUrl(url)
    .then(data :…  => {
        if (data) {
            //console.log('Fetched data:', data);
            //add subscriptions to the dropdown as they populate in the results
            const filters :string[]  = Object.keys(data) //create urls based localhost:8080/filter
            populateDropdown(filters);
```

Figure 32: The code retrieves data from the /data endpoint and populates the drop down based on the data available for each of the subscriptions

The data collected and analyzed is represented in a graph and the messages in each of the outlier buckets are listed. The graph showcases the counts of messages in each bucket and highlights the outliers in red for ease of use as shown in Figure 33. The x-axis displays the timestamp of each bucket. The y-axis is the number of messages in each of the buckets. Each

blue dot is the specific count in the given data set of frequencies and the red dots have been determined to be outliers. The data points are connected by lines to make it easier for the user to spot changes and patterns in the data.

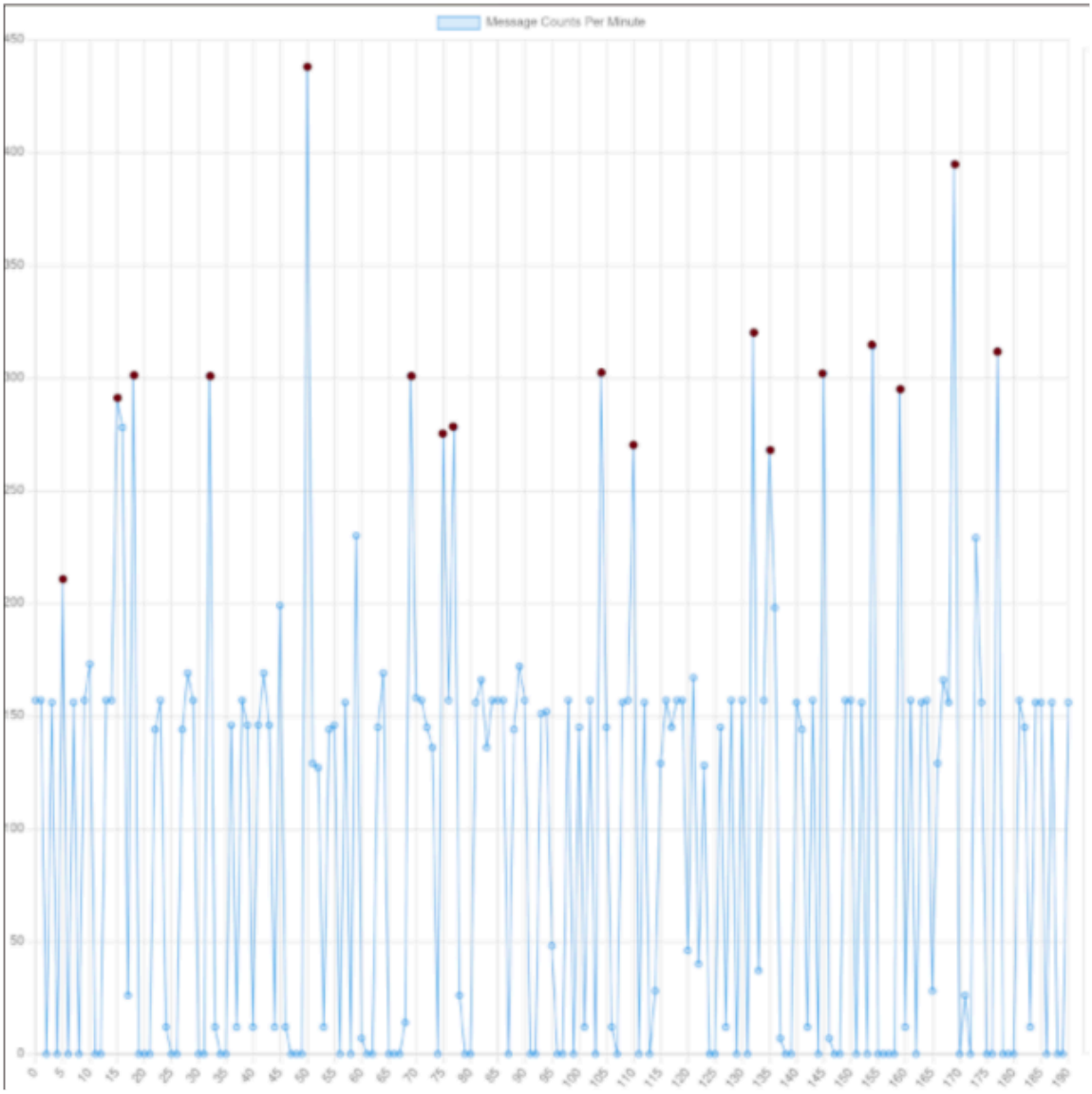


Figure 33: The graph displaying the number of messages per each minute monitored.

The data for frequencies is being collected from the /data endpoint every three seconds and updates the graph by redrawing it with the data points available as shown in Figure 34.

```
//selectedUrl is the specific filter/subscription
const url : string  = getEndpointForSub(selectedUrl);
const outlierUrl : string  = getOutliersEndpoint(selectedUrl);
//console.log(url);

fetchByUrl(url)
    .then(data => {
        if (data) {
            //console.log(filters);
            const counts : unknown[]  = Object.values(data);
            //console.log(selectedUrl);
            addData(chart, counts);
            addMsgs(outlierUrl);
```

Figure 34:The code retrieves data for each of the subscriptions and plots it for the user display

The chart is created using the chart.js library [10] with the following modifications to plot the given array of frequencies as shown in Figure 35 .

```
const ctx = document.getElementById( elementId: 'myChart').getContext('2d');
ctx.width = window.innerWidth * 0.5; // 50% of the viewport width
ctx.height = window.innerHeight; // 100% of the viewport height
const chart :An  = new Chart(ctx,  e: {
    type: 'line',
    data: {
        labels: [],
        datasets: [{
            label: 'Message Counts Per Minute',
            data: [],
            backgroundColor: 'rgba(54, 162, 235, 0.2)',
            borderColor: 'rgba(54, 162, 235, 1)',
            borderWidth: 1
        }]
    },
    options: {
        responsive: true,
        maintainAspectRatio: true,
        aspectRatio: 1,
        scales: {
            y: {
                beginAtZero: true
            }
        }
    }
});
```

Figure 35: The chart description and settings to get the best possible output

Similar to the frequencies data points, the outlier messages are collected from the /subscriptions data point and displayed as a scrollable box.

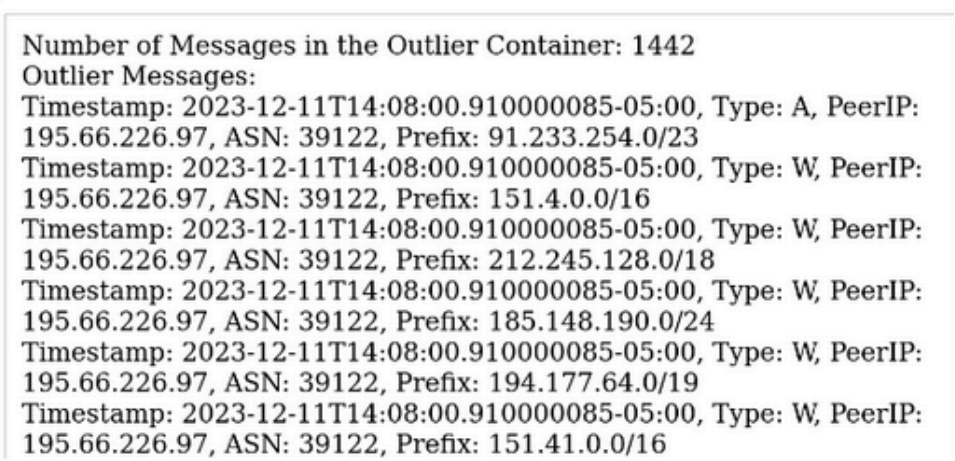


Figure 36:The list of messages displayed on the front end containing all the parsed information

The messages are retrieved from the endpoint in the following manner described in Figure 37. The messages for each outlier are encoded as a json on the endpoint, then the json is parsed and the values are mapped into strings based on their field names. Those field names correspond to the attributes of BGP Message structure.

```
//getting outliers from the outliers endpoint
fetchByUrl(outlierUrl)
    .then(data => {
        let textboxContent : string  = '';
        //go through all the timestamps and only get the message contents
        // Iterate through each timestamp (assuming each timestamp is a key in the JSON object)
        console.log(data);
        for (const timestamp in data) {
            if (data.hasOwnProperty(timestamp)) {
                const dataForTimestamp = data[timestamp];
                const counts = dataForTimestamp.Count;
                const messages = dataForTimestamp.Msgs.map(msg => {
                    return `Timestamp: ${msg.Timestamp}, Type: ${msg.BGPMessageType}, PeerIP: ${msg.PeerIP}, ASN: ${msg.PeerASN}
                }).join( separator: '\n');

                // Append the information to the textbox content
                textboxContent += `Number of Messages in the Outlier Container: ${counts}\n Outlier Messages:\n${messages}\n\n`;
                const msgsContainer : HTMLElement  = document.getElementById( elementId: 'msgs');
                // Set the value of the textbox with the extracted data
                msgsContainer.innerText = textboxContent;
```

Figure 37: Fetch data for the outliers for each subscription and format it for the textbox display

5.0 EVALUATION

In this section, we evaluate our tool MIRA by looking closely at the results of our anomaly detection algorithms.

5.1 Algorithm Performance Comparison

After running both of the algorithms against smaller datasets (10 or less points) and larger datasets (30+ points) some differences in outliers detected are as follows: ShakeAlert is better at determining local outliers, outliers relative to a subset of the whole dataset, while BLT MAD is better at determining global outliers, outliers relative to the whole dataset.

For smaller datasets with relatively extreme values, ShakeAlert was less sensitive to outliers than BLT MAD. For instance, consider the array [1,1,100, 1, 5, 300] where the ShakeAlert implementation with sensitivity parameter of 3 and BLT MAD implementation with the parameter of 3 have determined outliers displayed in Figure 38.

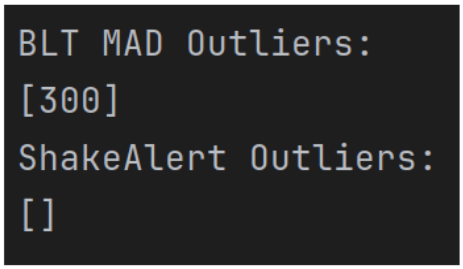


Figure 38: BLT MAD and ShakeAlert outliers determined on a smaller dataset

In this particular example with a smaller dataset, ShakeAlert is not able to detect any local or global outliers due to its nature of similarity. BLT MAD, on the other hand, determines the global outlier of 300 on this data set.

For larger datasets, BLT MAD tends to determine less outliers than ShakeAlert with a focus on highlighting the global outliers. For instance, consider the dataset displayed in Figure 39. Based on the data set above of 360 points where the sensitivity parameters for ShakeAlert and BLT MAD were 5 and 10 respectively, outliers displayed in Figure 40 were determined.

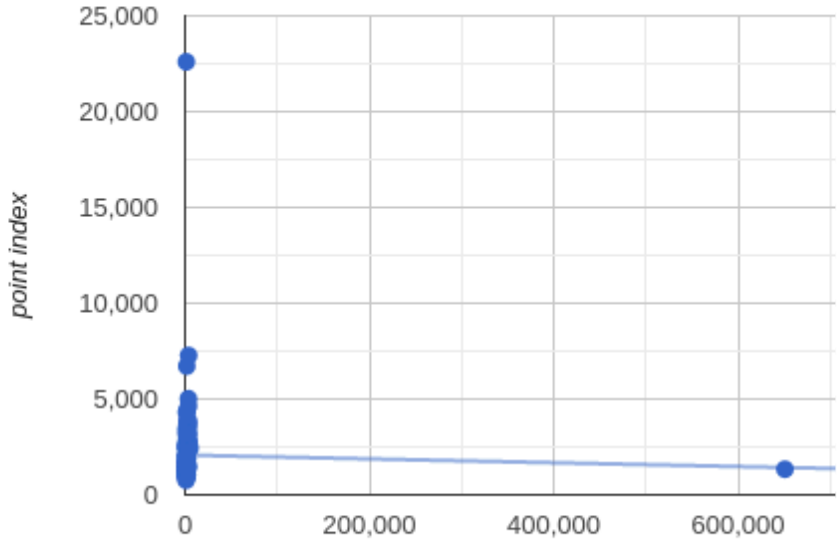


Figure 39: Graphical representation of static data containing 360 minutes of BGP message update counts

```
BLT MAD Outliers:
[650958]
ShakeAlert Outliers:
[22601 650958]
```

Figure 40: BLT MAD and ShakeAlert outliers determined on a larger dataset

Notice that BLT MAD was less sensitive to the outliers than ShakeAlert on the larger data set, determining only the global outlier and ignoring the local one.

Such differences in performance justify the simultaneous use of the two algorithms in our tool since using both provides a clearer picture of potential anomalies for the user to investigate further. Moreover, the sensitivity parameters in both of the algorithms control the number of outliers detected.

5.2 Linear Regression for Parameter Optimization for BLT MAD

The parameter optimization model was used to determine the sensitivity parameter for each given data set in an attempt to maximize the quality of outliers detected. The training and testing data was created by producing arrays of frequencies from static data. To investigate the effects of the quantity of data, the following three window sizes were used: 5, 15, 360.

The frequencies produced for each of the three window sizes were then used to calculate the mean and the MAD for each array. Then, the optimal sensitivity parameters were determined by calculating the maximum parameter value which produces the minimum required output. In this case, minimum required output was defined as the subset of each of the arrays larger than the 97th percentile of the given array. Such a minimum requirement allows one to focus on determining global outliers of the dataset but it may not be optimal for local outliers.

The accuracy of each model was calculated by finding the fraction of predicted values within 15% error of the expected values which accounts for the interval of values for the sensitivity parameter resulting in the same outliers detected.

The resulting models for each of the window sizes are displayed in Figure 41. The x-axis represents the ratio between the median and mean average distance as those are the two metrics

determining the outliers determined by BLT MAD. The y-axis is the predicted value for each sensitivity parameter.

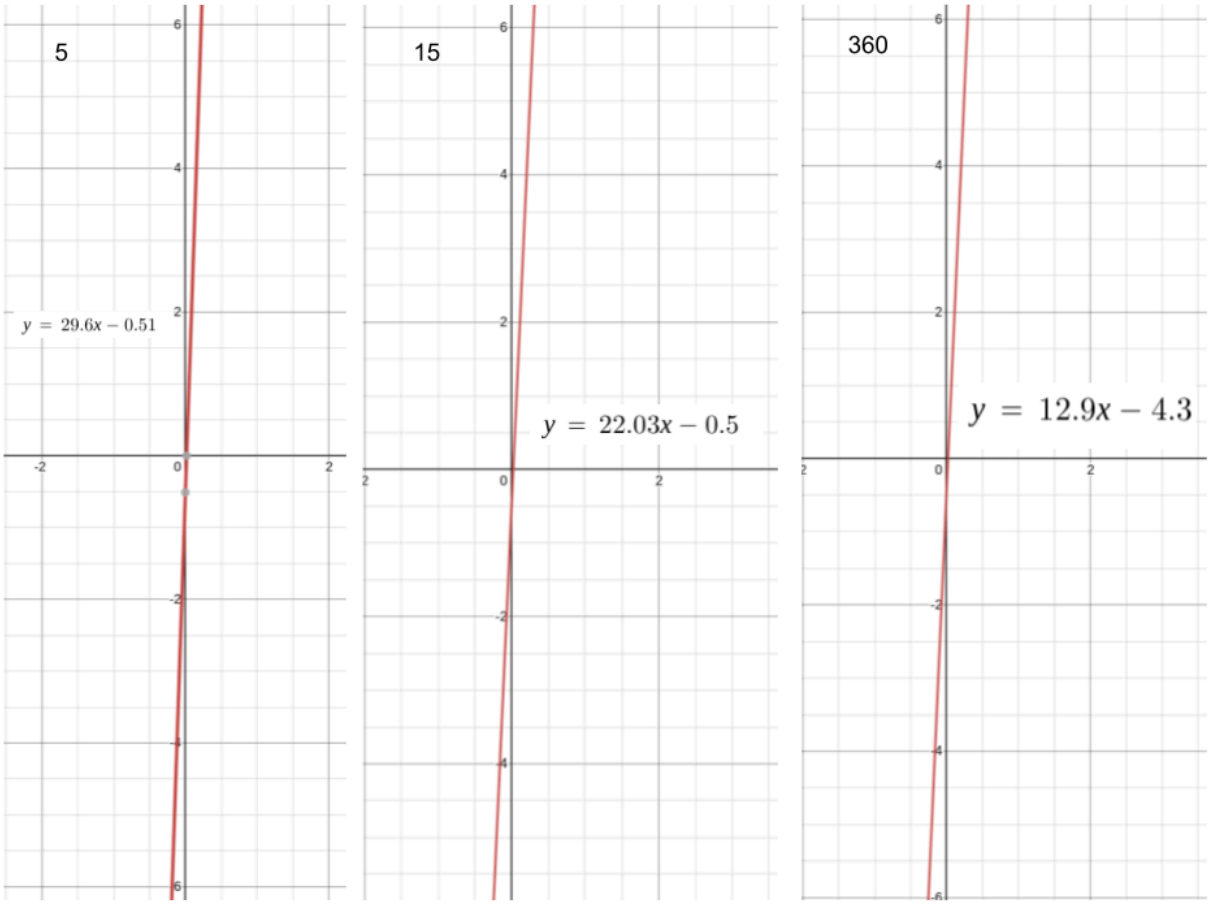


Figure 41: Graphical representation of the three linear models obtained for sensitivity parameter prediction for window size 5,15,360

The accuracy metrics for each of the window sizes are displayed in Figure 42.

Window Size	Predictions within 15% error
5	17%
15	27%
360	55%

Figure 42: Prediction metrics for window sizes 5,15,360

Since the model was chosen using the largest window size and since 360 is the default window size, that model was optimal for implementation in MIRA. Notice that for all of the models the slope is significantly larger than the intercept. Moreover, the models' accuracy can be improved by normalizing the data to remove bias. However, normalizing the input data also normalizes the predicted sensitivity parameter values meaning that an additional calculation is needed to convert the predicted value to match the non-normalized data inputs. Since those calculations can be computationally expensive for larger data sets, the normalized models were not implemented in MIRA.

6.0 FUTURE WORK

There are several ways to further improve MIRA in the future.

6.1 Classification of Anomalies

One significant improvement to MIRA would be the classification of anomalies once detected and reported. Currently, MIRA identifies anomalies in BGP update streams, offering a user interface that allows users to select an anomaly and view the associated BGP messages with the event. Users can then search for a pattern amongst the messages to understand what kind of event occurred. However, this relies on users having deep knowledge of BGP message events. An additional feature could be implemented to automate classifying the type of anomaly, such that MIRA both identifies an event and classifies the event using the BGP messages corresponding to the event. This has been explored by Kitabatake, Fontugne, and Esaki in their paper describing a BGP Update Labeling tool, where they used BGP message attributes in events to identify seventeen different BGP event types [4]. A similar implementation could be integrated with MIRA.

6.2 RouteViews Live Integration

MIRA currently parses live data from RIS Live, a service provided by RIPE RIS. RIS Live provided an abundance of documentation as well as a simple websocket interface that allowed for easy implementation and connection to RIS Live. RouteViews is a separate project collecting BGP routing updates from different route collectors around the globe. In the future, implementing a RouteViews Live listening stream for analysis may allow users to identify

different BGP events in real-time. Overall, having as many public vantage points of the global routing landscape should be useful for identifying events and their sources with higher confidence.

6.3 Algorithm Optimization

The outlier detection algorithm sensitivity parameters are dependent on the variability in a given time window thus setting a constant sensitivity parameter for all the analysis might not be the most effective. An optimization model may help to determine an appropriate parameter for each given dataset. Since the linear optimization model that we implement relies on using the sub-set of points larger than the 97th percentile of the dataset it may not accurately determine all of the relevant anomalies. Thus, creating another model that uses a different minimum required output to calculate the optimal parameter for training data would allow further optimization of outliers detected. Moreover, the optimization model is not self-learning meaning that the predictions do not improve over time. Future work may include adding an option for the user to indicate whether or not the outliers detected were reasonable and then feed that data back into the model as training data. In such a way, the new training data could avoid the issues caused by focusing on detecting the subset larger than the 97th percentile and instead reflect more pertinent outliers.

In addition, other types of outlier detection algorithms can be implemented to get a more complete picture of the given data set and gain insights that may have been missed by the BLT MAD and ShakeAlert algorithms.

7.0 CONCLUSION

The goal of CDNs like Fastly is to distribute traffic across the Internet to improve reliability and load distribution. CDNs rely on optimal Internet routing to balance traffic, thus identifying significant changes in the external routing posture is critical to their operations. One example of this is when large influxes of BGP updates occur during a short period of time, causing significant routing changes.

To address this, this project created MIRA, a tool that collects real-time BGP routing data and identifies when anomalous events occur. MIRA uses public BGP data and keeps track of the number of BGP updates messages occurring per minute. The tool offers a simple and customizable visualization for each user-specified subscription by graphing the number of incoming messages per minute, determining anomalies, and displaying each message associated with the anomalies. Engineers at Fastly can utilize this tool to understand when significant changes to Fastly's external routing posture occur and use the visualizations to better understand the root causes to then resolve the issue.

We evaluated our tool's functionality through extensive code reviews and application testing. Code reviews were completed for each of the pull requests we made, ensuring that our code was both efficient and well documented. Most of our testing was done by plugging in historical data of known BGP events and tuning our algorithms to verify that our program worked correctly on known data. We also made use of linear regression and other statistical models to fine tune the optimization parameters for one of our algorithms to provide more accurate results based on the type of data.

In summary, MIRA detects anomalies in BGP data with the potential to be a valuable tool

for engineers at Fastly allowing for optimization of their CDN.

References

[1] Cloudflare. (n.d.). *What is Routing? IP Routing*. Cloudflare. Retrieved December 6, 2023, from https://www.cloudflare.com/learning/network-layer/what-is-routing/

[2] Burke, J. (2023, June 1). *BGP (Border Gateway Protocol)*. TechTarget. Retrieved December 7, 2023, from https://www.techtarget.com/searchnetworking/definition/BGP-Border-Gateway-Protocol

[3] RouteViews. (n.d.). *About*. RouteViews. Retrieved December 8, 2023, from https://www.routeviews.org/routeviews/index.php/about/

[4] *Routing Information Service (RIS)*. RIPE Network Coordination Centre. (2015, March 4). Retrieved December 8, 2023, from https://www.ripe.net/analyse/internet-measurements/routing-information-service-ris

[5] *Ripe-NCC/bgpdump: Utility and C Library for Parsing MRT files*. GitHub. (n.d.). Retrieved December 11, 2023, from https://github.com/RIPE-NCC/bgpdump

[6] *Routing Information Service Live (RIS Live)*. RIPE Network Coordination Centre. (n.d.). Retrieved December 11, 2023, from https://ris-live.ripe.net/

[7] Flores, M. (2022, October 25). *Detecting waves with ShakeAlert*. Edgio. Retrieved December 12, 2023, from https://edg.io/de/resources/technical-article/detecting-waves-with-shakealert/

[8] Kitabatake, T., Fontugne, R., & Esaki, H. (2018). BLT: A taxonomy and classification tool for mining BGP update messages. *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. Retrieved December 12, 2023, from https://doi.org/10.1109/infcomw.2018.8406955

[9] *HTTP*. http package - net/http - Go Packages. (n.d.). Retrieved December 14, 2023, from https://pkg.go.dev/net/http

[10] Chart.js. (n.d.). Retrieved December 14, 2023, from https://www.chartjs.org/