BGP Route Leaks Analysis

Benjamin Wijchers
Faculty of Exact Sciences, department of Computer Science
Vrije Universiteit Amsterdam

December 3, 2014

Supervisors:
Dr. Benno Overeinder (NLnetLabs)
Dr. Paola Grosso (Universiteit van Amsterdam)

Second reader:
Dr. Spyros Voulgaris (Vrije Universiteit Amsterdam)

Abstract

The Internet is a network of networks, where reachability information is announced between the networks to establish connectivity between them. The protocol over which this reachability information is distributed, BGP-4, has several security vulnerabilities. Most of these vulnerabilities will be covered by the introduction of improvements such as RPKI and BGPsec, but a partially unsolved vulnerability are so-called *route leaks*.

A route leak is a violation of the policies between the networks involved. It occurs when a network B, announces a route learned from network A to another network C, even though the policies between the three involved networks prescribe that this route should not have been send from network B to C. Non-disclosure agreements about the nature of the relationships between networks make it hard to distinguish route leaks from regular announcements.

In this project, recent relation inferences from CAIDA have been used to detect possible route leaks in publicly available BGP data. These potential route leaks have been further investigated on their duration, the type of violation, and the type and origin of network that caused the leak-detection. Most detected possible route leaks had negligible durations. The ones with a longer life-time mostly involved special relationships between networks not currently inferred by relationship datasets. Other possible route leaks detected require more information about the networks involved to properly analyse the situation.

Contents

Chapter 1

Introduction

People, businesses and countries are dependent on the Internet. The Internet has shown tremendous scalability, stability and robustness. However, there are still a number of open scalability and security problems. The Internet is a network of networks, where direct and indirect connections realize global reachability. How traffic flows between two or more networks is determined by the Border Gateway Protocol [36]. BGP is a routing protocol that allows implicit expression of policies. New enhancements to the protocol are made to provide security, but not to enforce policies. This thesis studies the policy violations, how they appear, and what their frequency and impact is amongst other characteristics.

1.1 Problem Description

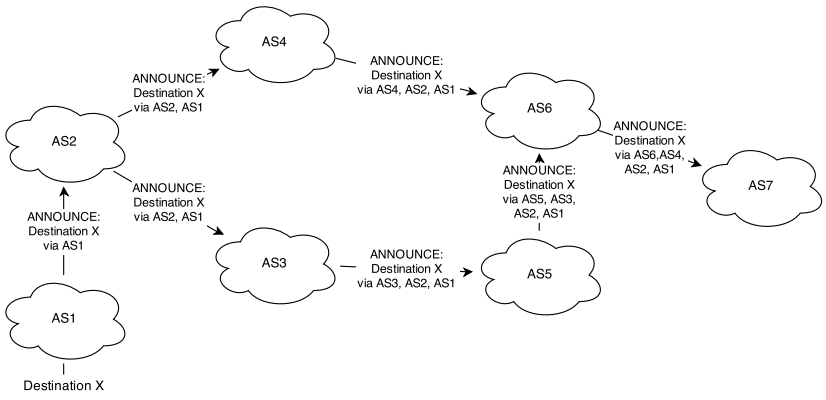


Figure 1.1: AS1 announces reachability to destination X, this announcement is spread by other ASes. AS6 receives two announces to the same destination and chooses one according to local policy and distributes only this one further.

The Border Gateway Protocol is a peer-to-peer protocol used between connected networks to propagate routes between each other. When a network has received different paths to a single destination, certain route characteristics will be considered to choose whether or not the newly received route should be accepted as the new best route to that destination. Depending on the policy between networks, new routes received should or should not be announced to certain other networks as well.

A major shortcoming to BGP is its lack of security within the protocol [28]. A neighbour network can adjust the perceived route characteristics to make this route seem more optimal to manipulate traffic for that destination to be re-directed over itself instead of over other networks. The protocol currently does not provide means to differentiate a correct route from a route that has been tampered with.

Recently more attention to security and trust has started some work in the IETF. RPKI[17] and route origin validation are currently being rolled out, which will allow networks to validate if the destination announced by a network as belonging within it, does actually belong to that network. Another proposed enhancement to the protocol is called BGPsec[18], which is currently under discussion within the IETF. BGPsec will allow networks to validate the attribute that indicates which hops the announcement has already traversed, which can also be manipulated to attempt to create man-in-the-middle attacks in the original BGP-4 protocol.

However, neither of the proposed enhancements to the protocol will secure the routing infrastructure against policy violations [24]. This is when network B announces a route learned from network A to another network C, even though the policies between the three involved networks prescribe that the route should not be propagated from B to C. Since network C may not be aware of the policies between networks A and B, and the announcement is valid according to BGPsec, RPKI as well as the policy between networks B and C, network C can still accept the route even though this introduces a violation according to the policy between networks A and B.

1.2 Motivation

When a policy violation occurs, this will have certain negative influences on the way traffic is routed. When a network leaks a route unintentionally, this will usually result in more traffic being routed over the leaking network. Since the policies with that network do not define that this network should handle this extra traffic, the network will not get financially compensated to handle this extra traffic, even though the network itself will need to spend money to

handle the traffic, resulting in obviously negative financial consequences for the leaking network. Since the network usually does not handle the increased amount of traffic, it is also likely that the hardware installed to handle traffic is insufficient to cope with the additional data to handle and traffic via that network will be transferred slower than when larger networks would handle it.

However, a route leak may be created intentionally as well to intercept traffic between two networks that are both connected to the malicious network. Since the route announcement will be originated from a non-provider network, the route will get additional preference over a legitimate provider route (as providers ask money for transit, while clients pay to do so). Since traffic now is routed over the malicious network, this network can inspect the data being transferred between its two neighbours and possibly manipulate or drop it.

1.3 Contribution / Research question

"Are there ways to measure / determine route leaks in a generic way, can we analyse the (root) cause of the leak, and can we categorize these incidents in classes?" is the main question that will be treated in this article. Since there are multiple reasons why a route leak occurs, one more alarming than the other, it may be interesting to see if there are means to distinct one type of route leak from the other. As part of this research question we will attempt to answer the following sub-questions:

- What part of all routes announced can be considered a route leak?
- How long does it take until the path of the route leaks is withdrawn from service?
- Is the amount of active route leaks over time growing, shrinking or staying the same?
- Are certain (groups of) ASes responsible for most of the route leaks or are they distributed over all ASes?
- Can we determine the causes of the route leaks (e.g. are they intentional or a misconfiguration)?

1.4 Approach

In order to answer these questions several steps have been taken. First, related work has been looked at to investigate upon previous attempts at finding route leaks, as well as approaches at finding relationships between networks, which will be discussed in Chapter 2.

Next, a design was made for an application that will help with the analysis of this project. This application first gathers publicly available BGP traffic dumps which will be used for analysis. The tool will then automatically analyse the dump files using an inferred relational database to detect and store route leaks found in a local database, along with several leak properties. Furthermore, the application will also be able to query the database to generate statistics from the information gathered and to visualize the results to unveil various route leak characteristics.

The design of this application will be discussed in Chapter 3, while the implementation will be discussed in Chapter 4. Some noticeable recurring cases have been studied further manually and the results of this further analysis have been written in Chapter 5.

Chapter 2

Related Work

In the past, many route leaks [8, 7, 33, 34, 16, 5] have been discussed. Many of them could be detected because certain properties of the routes announced were manipulated such as the prefix origin, which can be detected by the prefix owner by using services such as BGPMon[43]. However, some of them [33, 16] do not modify the routing information and still leak the route they legitimately received. Because of the lack of knowledge of relations between ASes it has been hard for outsiders to detect the last type of route leaks. In this chapter various approaches at obtaining this knowledge of relations between ASes and previous attempts at finding route leaks will be discussed.

2.1 Introduction to the Border Gateway Protocol

BGP-4 is a peer-to-peer protocol used to propagate routes between different Autonomous Systems (ASes). An AS is defined as "a connected group of one or more IP prefixes run by one or more network operators which has a SINGLE and CLEARLY DEFINED routing policy." [15] Every AS has its own unique AS Number registered that is used within BGP to identify the AS.

When two ASes have established a BGP connection, routes will be shared using update messages. Update messages announcing a path to a destination contain one or more Network Layer Reachability Information (NLRI) fields to define which IP prefixes can be reached using that path and several other path attributes. One important BGP-4 field is the AS_PATH, which contains an AS_SEQUENCE attribute where every AS propagating the update further prepends it ASN to, making it an ordered list of all ASes traversed. The main goal of AS_PATH attribute is to prevent cycles being formed in a route, but

it is also used to determine how many hops there are to reach the destination propagated. An additional benefit of the AS_PATH is that it allows us to determine what path the announcement to a certain destination has traversed before arriving at the vantage point for the BGP traffic dumps we will be investigating.

2.2 About the Valley-Free Rule

The field of inferring relations between ASes was started by Gao [9] in "On inferring autonomous system relationships in the Internet". In this article Gao defined the valley-free rule and relations were derived making use of this rule. In this section it will be explained what the valley-free rule is and why it is usually followed. There are however also several reasons why networks choose to violate the valley free rule, several known reasons to do so will also be discussed later on.

2.2.1 Definition

According to the valley-free rule, the relation between two different ASes can either be customer-to-provider, peer-to-peer or sibling-to-sibling. In the case of a customer-to-provider relation, the client pays the provider to reach parts of the Internet that are outside of the client's own clients. When two ASes establish a peer-to-peer relation, the two ASes agree to provide transit to each other's clients free of charge. By doing this, they eliminate the need of both paying a provider for traffic targeted towards the peer's clients. A sibling-to-sibling relation is a special type of relation between two ASes which share the same organization behind it. Since both ASes have the same owner, they share all of each other's routes (contrary to peer-to-peer where only client routes are shared).

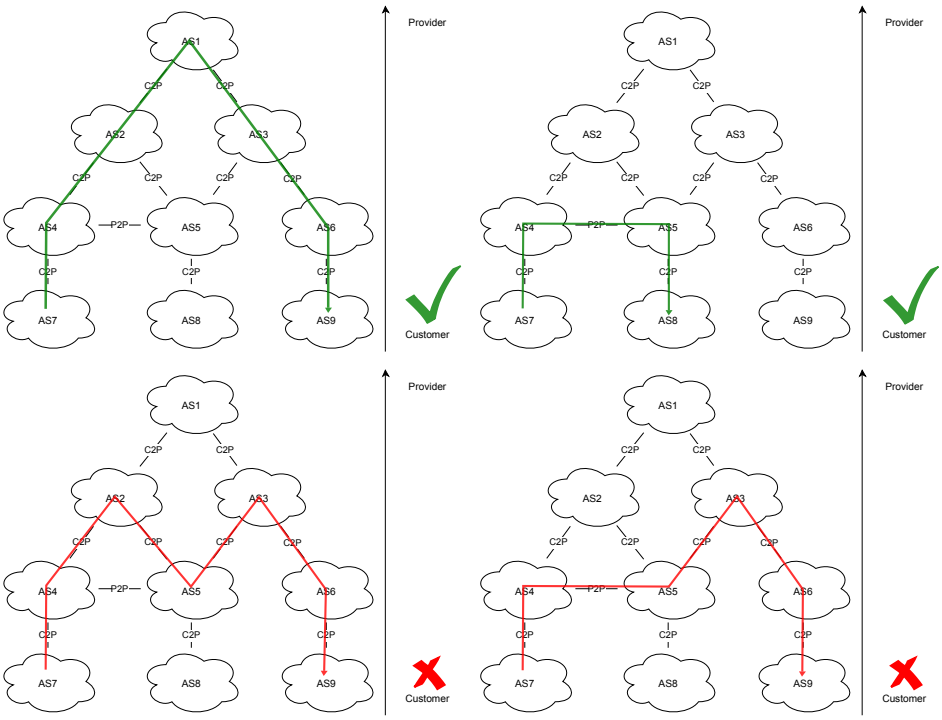


Figure 2.1: Examples of (non) valley-free paths

The valley-free rule states that any AS path announced should start with any amount of client-to-provider edges, followed by either one or no peer-to-peer edges, followed by any amount of provider-to-client edges. Sibling-to-sibling edges may appear anywhere in this rule and thus will not cause a valley. This rule is usually valid, as any violation of it will result in an AS paying for the transit of another AS.

2.2.2 Reasons for valley free violations

Giotsas and Zhou [13] mentioned that valley-free violations were formerly considered to be a mistake which were usually caused by routing configuration errors. They discovered however that up to 50% of the valley-free violations are actually intentional. They have found that the majority of the prevalent valley-free violating routes propagated are found to be caused by research/educational or IXP ASes. The valleys created between research/educational ASes were found to likely be caused by forms of indirect peering, where the two ASes want to establish a peer-to-peer relation, but have no common Point-of-Presence and thus use intermediate ASes between them.

Since neither of the ASes is the other's client and paths will be created both ways, a valley will occur within those paths. IXP ASes can cause valley-free violations similarly as they work as a point where indirect peering relations are also formed. When the IXP AS is then injected between the two peering ASes a valley is also inevitable for the same reason mentioned before.

Giotsas and Zhou [12] noted that announcements of IPv6 prefixes have relatively more valley-free violations than IPv4. It is suggested this should be necessary to be able to maintain global reachability within the IPv6 Internet as not all ASes currently support this protocol. According to the same article, some ASes are also found to have different relations for IPv4 and IPv6 traffic.

Several other possible reasons for ASes violating the valley-free rules have been speculated by Mazloum et al. [23, p. 8]. One of them would be that a certain AS could have insufficient bandwidth to route traffic over that link and that therefore a non-valley-free route is preferred over that one. The same can be done when a route is geographically further away. A reason why route leaks may be detected incorrectly, mentioned in [23], is that the relationships inferred by CAIDA[39] are derived with an understanding of complex/hybrid relationships [20, p. 8], but still output only peer-to-peer and client-to-peer relations. Thus, if two ASes have a hybrid relation, where in certain conditions their relation would be peer-to-peer and in other situations client-to-peer, one of those routes will likely be seen as a valley-free violation. Mazloum et al. also mention that, although the CAIDA AS relations has a high accuracy in validated results, there still can be relations that are incorrectly derived and that are therefore causing wrongly detected valleys.

2.3 CAIDA's AS-Rank algorithm

In the article "AS Relationships, Customer Cones, and Validation" Luckie et al. [20] describe the methodology used for the inference of the relation data used by CAIDA's AS Relationships Dataset [39]. Contrary to other approaches, the algorithm used for the CAIDA relationships does not assume that the AS paths they investigate are valley free. Instead it relies only on the following three assumptions:

- "there is a clique of large transit providers at the top of the hierarchy;"
- "most customers enter into a transit agreement to be globally reachable"
- "cycles of c2p links (e.g., where ASes A, B, and C are inferred to be customers of B, C, and A respectively) should not exist for routing to converge"

The algorithm uses the paths found in publicly available BGP routing data and starts by filtering out incorrect AS paths (e.g. containing reserved AS numbers or cycles). Then ASes are sorted by their transit degree. This is the number of different ASes from which prefixes are forwarded by an AS which should be the number of ASes for which that AS provides transit.

Next, the clique of the top of the topology is formed by first taking the 10 biggest ASes according to their transit degree and then determining which other ASes are connected to all ASes in that clique, but do not receive transit from them (have paths containing 3 clique members if that AS would be in the clique).

After the clique is formed triplets of 3 neighbouring ASes in the AS path are iterated. When the relation between the first two ASes in the triplets is either P2P or P2C and the relation between the last two ASes in the triplet was not yet known it will be inferred to P2C as otherwise the middle AS would be leaking a provider or peer route. Initially only the P2P relations from the clique will be known, but when a P2C relation is inferred in the previous iteration step, this will be used in next iterations as well allowing 90% of the relations to be inferred using that step.

After this step additional steps will be made to infer C2P relations from vantage points inferred to be announcing no provider routes, to infer C2P relationships for ASes where the customer has a larger transit degree, inferring customers for provider-less (non-clique) ASes and infer stub ASes as customers of clique ASes.

When all previous steps have been executed, all ASes are traversed once more to find triplets with two consecutive unknown relationships. For the first two ASes in the triplet an alternative third AS will be searched which is a provider of the middle AS. When such a triplet is found, the first AS in the triplet will be inferred as being a customer of the middle AS as any other inferred relation results in a route leak in the found triplet. When this alternative triplet is not found another triplet will be searched where the first two ASes are in reverse order in the path, with another AS in front of it with also an unknown relation. If this AS is found the first AS from the original triplet will be inferred as a customer of the middle AS from the original triplet.

When all previous steps have been taken any links that have no inferred relation yet will be assigned a P2P relationship. Because the previous step attempted to infer relations for two consecutive previously unknown relationships, the amount of consecutive P2P-P2P relations in AS paths should be minimal.

The algorithm's accuracy was validated using 34.6% of the data which showed that 99.6% of the customer-to-provider edges and 98.7% of the peer-

to-peer edges were found to be accurate.

The existence of hybrid relations is discussed and techniques are described to mitigate some of the effects of them for calculating the customer cone data. However, since the relationships data is used by our application, which does not differentiate hybrid relations from "normal" relations, this will still influence our data.

2.3.1 Siblings inference

The inference of siblings relationships was left as future work in the original article. Later the article *Mapping Autonomous Systems to Organizations: CAIDA's Inference Methodology* [6] was published along with the inferences of a mapping from AS number to organisation. This data can be used as a siblings inference as ASes belonging to the same organization are considered siblings.

The inference of the AS to Organization data uses WHOIS databases from five Regional Internet Registries and two National Internet Registries and detects similar entries for different AS numbers to identify them as ASes from the same organization. The structure of WHOIS databases is undocumented and it varies per database. On top of that, the databases are updated manually and the data inserted is often outdated or incorrect.

Therefore, the first step in the algorithm is to "prepare the data for uniform inter-database analysis". Here the data is normalized and non-relevant data is filtered out. The second step is to group data that contains the same data in various fields (e.g. email, name, etc.). Here, generic values are not used and the fields being used are tested to result in maximizing the amount of true positives while minimizing the number of false positives. The final step is validating the results with ground-truth data for various organizations.

Although the number of false positives produced is arguably significant, it is unlikely this will pose a problem for the purpose we will be using the data. For a valley to be incorrectly interpreted as a non-valley because of the siblings data, the following condition should be met: An AS that leaks a route should be incorrectly inferred as a sibling with the AS from which the route is leaked. Therefore the two ASes inferred to be siblings should share a significant portion of WHOIS data and are neighbour ASes. This scenario seems unlikely, except for when an organization is split up and did not update the WHOIS data. Another possible, but even more unlikely, scenario is that an AS intentionally modified the WHOIS data to resemble the other AS, so an intentionally formed route leak looks more legit. This would require the leaker to adjust publicly available information in suspicious ways making the malicious intentions too obvious.

2.4 UCLA's relation algorithm

Another publicly available AS relations dataset considered for use with this project is the one from UCLA[44]. In "Quantifying the completeness of the observed internet AS-level structure" [32] the algorithm which is used to generate this dataset is described. For this algorithm the following assumptions are made:

- "the set of Tier-1 ASes are already known"
- "monitors at the top of the routing hierarchy (i.e. Tier-1 monitors) are able to reveal all the downstream provider-customer connectivity over time"
- routes follow a no-valley policy

The algorithm starts by assigning a peer-to-peer relation between every Tier-1 AS. After that, routes received by Tier-1 ASes are used to infer customer-to-provider relations for other ASes. When an AS is directly followed by a Tier-1 AS in an AS-Path, the relation can be either P2P or C2P. When an AS is directly followed by two consecutive Tier-1 ASes in the AS-Path, the relation between the AS and the Tier-1 AS linked to it will be C2P, as the link between the Tier-1 ASes was already P2P and Tier-1 ASes have no providers and having two consecutive P2P relations in an AS-Path forms a valley. Any AS that occurs further before first AS before the Tier-1 AS is inferred as a customer of the AS after it because, according to the valley free rule, the uphill path should only contain customer-to-provider links and when a Tier-1 AS is in the path, the whole path before it should be the uphill path. Customer-to-provider links should all automatically be discovered with this method, as every AS attempting to reach global connectivity should be connected using provider links to a Tier-1 AS. Peer-to-peer links will not be discovered with this method, as peers only share client routes and thus a client route received from a peer should not be send to the provider (or other peer) and will thus not be seen by a Tier-1 AS. Therefore, every link between two ASes that does not yet have a customer-to-provider inference will be inferred as being peer-to-peer at the end of the algorithm. The algorithm filters out routes that had a very short lifetime (less than 2 days) to decrease the number of incorrect inferences caused by BGP misconfigurations or hijacks.

2.5 Previous BGP Route Leaks project

2.5.1 Mauch

The first project known to us that automatically detects BGP route leaks in publicly available BGP traffic dump files was the project by Mauch [21]. Mauch created a Perl script which is able to identify valley-free violations within the publicly available traffic dumps and displays them onto the website. The script uses a few ground truth rules and only detects a valley when 3 or more major (Tier-1) networks appear in a single AS path [22]. The advantage of this method over that of using relations inferred by others is that those relations can contain a lot of incorrect information, while the small amount of major networks used, can be checked by a single person. The disadvantage is that all valley-free violations that do not involve those few major networks will not be detected. The script will automatically output the valley paths to his website with the time, prefix announced, AS path and information about which AS can be contacted or blamed, as well as a score value which indicates how many major networks were found in the AS path.

2.5.2 Vouteva and Overeinder

The project by Mauch lead to a follow-up project by Vouteva and Overeinder [41], which our project is loosely based on. This project was able to combine relation data derived by UCLA [32] with the BGP dump files retrieved from RIPE[38], Oregon[25] and WIDE[25] to detect valley-free violations. Apart from filtering the BGP messages with routes containing valleys, no further analysis on the valley routes is performed. Although our project was originally based on this project, it was decided not to build further upon the code from this project as the code was written for a very specific setup. The new project will be open source and needs to run on multiple setups.

2.6 Our Contribution

This project was originally started using the code from the Vouteva and Overeinder project. However, since new requirements for the project required too much of the code to be changed, the code was written from scratch instead. What has remained from the Vouteva and Overeinder project is the sources used for the BGP traffic dumps and the method to detect valleys in them. Also the programming language used (Python) has remained the same, although it now supports newer versions as well. Also, this project

uses the AS relations derived from CAIDA instead of UCLA and it uses MySQL instead of PostgreSQL. Furthermore, much functionality has been added that was not available in the project by Vouteva and Overeinder yet. The most prominent new feature is that our application allows for statistical analysis on the valleys apart from only detecting and storing valleys. Another enhancement is that it also detects when valleys stored are withdrawn from service by another update and stores duration information in the database. Also more context for the valleys found is stored, such as the ASes who form the leak and the location (country) of those ASes. On top of that, while the Vouteva and Overeinder project was built for a specific set-up, this project uses a more generic approach.

Chapter 3

Design

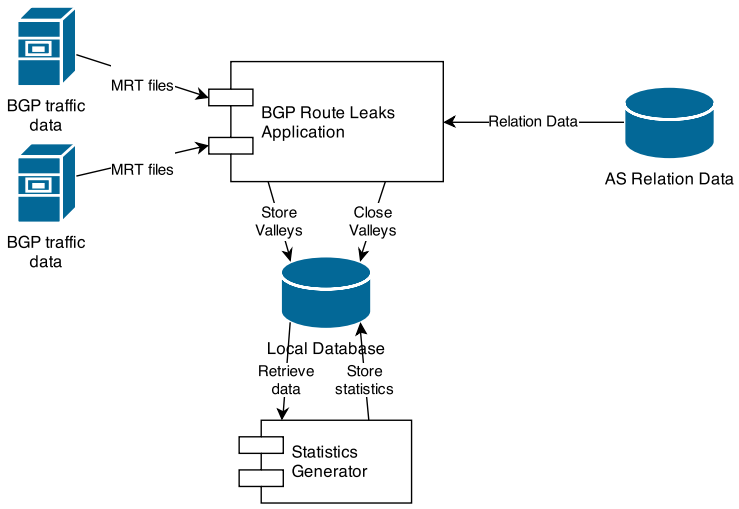


Figure 3.1: Design of the application

Figure 3.1 shows the design of the application. The application will need two forms of input: the BGP traffic dump data it will analyse and the relation data containing the relations between different ASes. The BGP Route Leaks Application will then parse the BGP traffic data using the relations data and store valleys found in the local database. Also, for every update in the BGP data, it will be checked if it will withdraw a valley that was previously added to the local database. The statistics generator will be activated when a certain range of valleys is added to the database and will query the local database in various way to generate statistics.

3.1 Valley detection algorithm

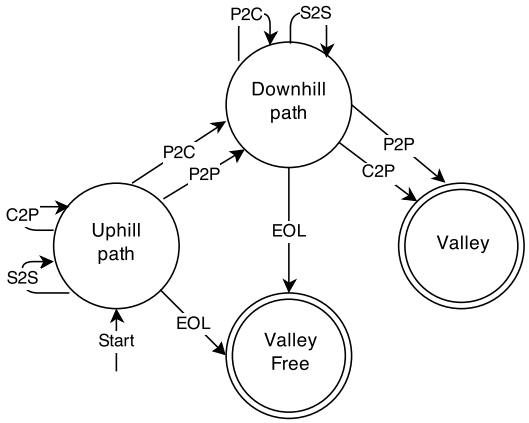


Figure 3.2: FSM for the valley detection algorithm

The valley detection algorithm is pretty straightforward. As mentioned in Section 2.2 a valley free path consists of any number of customer-to-provider (C2P) edges, followed by at most one peer-to-peer (P2P) edge, followed by any amount of provider-to-customer (P2C) edges. To detect a valley we therefore read the AS path of an update, reverse it (as the ASes are prepended to the AS path, the first one sending the message will be the last one in the sequence) and retrieve for every AS neighbour pair the relation between them. The list of relations will then be traversed until either a P2P relation, or a P2C relation is found. After either of these relations is found, the rest of the list will be traversed until a C2P or a P2P relation is found. If either of these relations is found, the update will be considered a valley. If the whole list is traversed before a valley is detected, the update will be considered valley free. Sibling relations have no effect in any part of the algorithm and when a relation between two ASes is not known, it will be handled the same way as a siblings relation.

3.1.1 Siblings

Since the relations file we used is combined with siblings data and we would like to see the effects of this combination, we have altered this algorithm slightly. First the relations which were found to be sibling-to-sibling relations are interpreted as if they were the originally inferred relationship (e.g. C2P, P2P or P2C). When a valley is found, the list will be parsed again, but now with the siblings relations ignored. When there is still a valley, the update

will be marked as a "real" valley, otherwise the update will be marked as a "non-real" valley.

3.1.2 Additional valley attributes

The valley detection algorithm we implemented also generates other valley attributes that are of interest and that are stored in the database. One such attribute is the *violation type* which contains the two relations that together form the valley (e.g. P2P-P2P). This attribute is created by storing the relation that caused the transition from Uphill path to Downhill path and the relation that caused the transition from Downhill path to Valley and combine them. When the relation type appeared to be a siblings relation according to the algorithm from the previous subsection 3.1.1, this information will also be included in the violation type without removing the original relation (e.g. sP2P-P2P indicates a "non-real" valley that is actually S2S-P2P, but had an initial inference of P2P-P2P).

Another attribute that is inferred by the valley detection algorithm is the *"leak triplet"*. The leak triplet contains the three ASes that together form the valley: the *leaked from* AS, which is the AS who sends the route to the leaker, the *leak AS* who causes the valley and the *leaked to* AS, to whom the leak AS sent the route. The *leaked from* AS is defined at the transition from Uphill path to Downhill path, while the *leak AS* and *leaked to* are set to the once involved at the transition from Downhill path to Valley.

3.2 Valley closing algorithm

According to the definition in RFC4271[36] there are three methods for an AS to announce that a previously announced update has been withdrawn from service:

- "the IP prefix that expresses the destination for a previously advertised route can be advertised in the WITHDRAWN ROUTES field in the UPDATE message, thus marking the associated route as being no longer available for use,"
- "a replacement route with the same NLRI can be advertised, or"
- "the BGP speaker connection can be closed, which implicitly removes all routes the pair of speakers had advertised to each other from service."

Therefore, we check for every update announced if there is a not-yet-closed valley in the database which was previously announced by the same source IP with the same announced prefix as this new update announces or withdraws. If this is the case this valley will then be "closed" by this new update. Also, when a BGP update containing a state change arrives and the previous state was ESTABLISHED, all valleys created by that source IP will be closed as well.

However, since our algorithm also changes the AS relation data monthly, there may be valleys in the database which are not detected as a valley when the new relation data is used. Therefore, when the AS relation data changes, all valleys that are in the database that are not yet "closed" will be re-checked with the new relation data to see if it still contains a valley. If this is not the case these valleys will be closed with the end time set to the last second before the beginning of the month were it was not detected as being a valley anymore.

3.3 Database Design

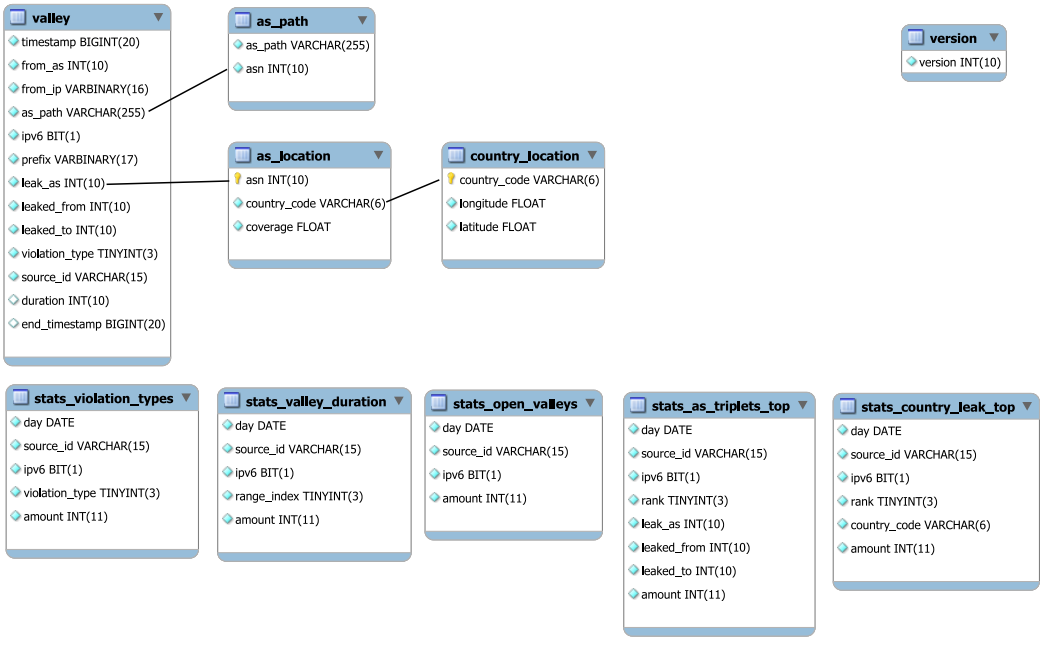


Figure 3.3: Design of the database

Figure 3.3 shows the design of the database. The main table here is the *valley* table, where all valleys found will be stored. The *as_path* table is added to

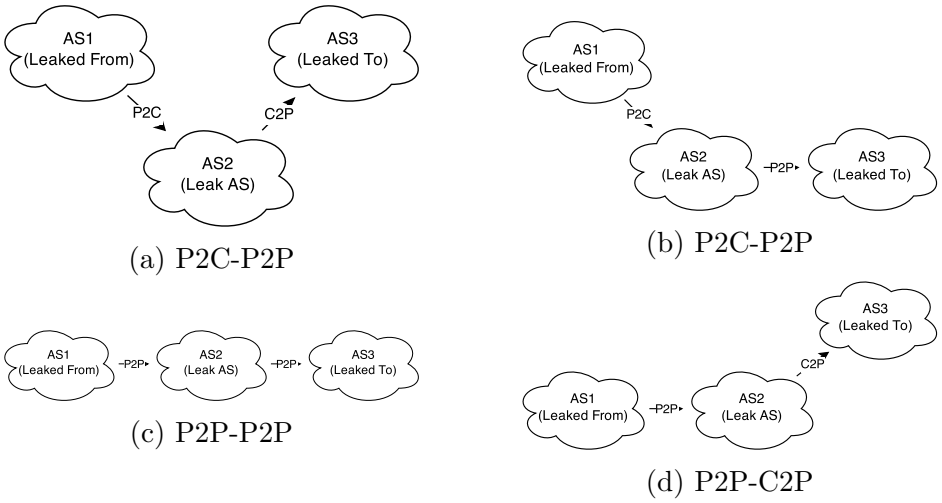
allow lookups in the table for valleys containing a specific AS number in the path. The *as_location* table is added to allow the *leak_as* to be linked to a specific country and the *country_location* allows to link that country to a certain longitude and latitude combination.

From every valley found, the following information will be added to the valley table in the database:

- *Timestamp*: The date and time of the valley announcement. As multiple announces are being made at the same second, and it is sometimes necessary to have knowledge about which happens first, it was decided to store the date and time in a timestamp field with millisecond accuracy.
- *From AS*: The AS number of the neighbour AS who propagated the update to the route collector
- *From IP*: The IP address of that neighbour
- *AS Path*: The ASPATH attribute of the announcement. The textual representation of the AS path is stored in the database, to make it possible to lookup individual ASes that are in that AS path, the *as_sequence* table is made, containing every combination of the *as_path* attribute with every *asn* that is in that *as_path*.
- *IPv6*: Indicates whether or not the announcement was for an IPv6 prefix
- *Prefix*: The prefix announced
- *Leak Triplet*: The three AS numbers that induce the valley in the path, consisting of:
  - *Leak AS*: The AS number of the AS who caused the valley within the AS path. The Leak AS is also a foreign key to the *as_location* table, so the geographical origin of the leaking AS can also be queried.
  - *Leaked From*: The AS number of the neighbour of the leak AS who propagated the route to the Leak AS
  - *Leaked To*: The AS number of the neighbour of the leak AS who received the route from the Leak AS, although he should not receive it according to the valley free rule

- *Violation Type*: The combination of the relations between Leaked From and Leak AS and between Leak AS and Leaked To. There are 4 types of relation combinations that cause a valley: P2C-C2P, P2C-P2P, P2P-P2P and P2P-C2P (see Figure 3.4). However, when a valley is reparsed with the siblings information and deemed not a "real" valley, it will still be stored, but with the siblings relation noted in the violation type (e.g. P2P-sP2P is a non-"real" valley).

- *Source ID*: The source of the route collector, e.g. RIPE. This allows to generate statistics per source to determine if the (location of) route collector influences the valleys perceived.



(a) P2C-P2P

(b) P2C-P2P

(c) P2P-P2P

(d) P2P-C2P

Figure 3.4: Violation types

On top of that, if an update withdraws a previously added valley from service, the End Timestamp will also be set to the time of the update that closed the valley. Furthermore, the duration will also be set to the difference between the End Timestamp and the Timestamp values to allow optimization of queries needing the duration value.

Since the amount of data that will be stored in the database is quite high and not all of the queries needed to display certain statistics can be fully indexed, several tables with pre-calculated statistics have been added as well. This also allows to store statistics for a longer time-period while older individual valley information is removed from the database to save disk space.

The tables *stats_violation_types*, *stats_valley_duration*, *stats_open_valleys*, *stats_as_triplets_top* and *stats_country_leak_top* are the statistics tables. These

tables will contain statistics for a single day and to get the statistics for a longer time period the values of multiple days can be summed together. Since multiple sources for BGP data are used, and sources may be added or removed while there is already data in the database, the statistics are also calculated for each source separately to avoid inconsistency when the sources change. Also to allow differentiation of statistics for IPv4 and IPv6, the statistics are kept separately per IP version as well. The last common field that all stats tables share is the amount field, which contains the number of valleys that share the conditions specified by the table and the table-fields.

In the stats_violation_types table, the amount of valleys detected that had a specific violation type. Therefore this table has the violation type that is counted as an extra field. The violation type is an integer that represents a specific violation type (e.g. 1=P2P-C2P, 2=P2P-P2P, etc.)

The stats_valley_duration table counts the valleys that had a duration between a certain range (0-2sec, 2-4sec, 4-8sec, etc.). Therefore this table has as extra field range_index indicating between which range the valleys were counted.

The stats_open_valleys table counts the total amount of valleys that have been announced before 08:00 that day and have not yet been withdrawn by other updates. Since the table contains counts for only one specific point of time at any given day, there are no additional fields for this table. The choice for 08:00 is arbitrary, a certain time had to be chosen. Since it counts all the updates that were announced before that time and withdrawn from service after that time and thus mainly counts valleys with longer durations, it is expected that the exact time chosen should not significantly influence the statistics.

The stats_as_triplets_top contains the most occurring AS triplets of that day. Therefore the AS triplet (leak_as, leaked_from and leaked_to fields) have fields. On top of that the rank (0 is most frequently occurring, higher is less occurrences) is also stored.

The stats_country_leak_top works similar to the stats_as_triplets_top table, but instead of the AS triplet fields it contains the country_code field.

Chapter 4

Implementation

4.1 Working of the application

When the main application starts, it will look into the database to find the last valley added for every data source. Then the updates dump file will be searched that contained this valley, which will be the first file to be parsed to look for valleys, to see if the program was previously shut down while in the middle of the process of adding valleys (to prevent inconsistency by missed valleys). The parsing of an updates dump file goes as follows:

- If the relation file is not initialized yet, or a new custom relations file needs to be selected (e.g. update is in new month), the relation file will be initialized in accordance to Section 4.5.
- If the updates file is not yet downloaded it will be downloaded from the server.
- The updates files are in MRT format [4] and will thus first be decoded using the libbgpdump library described in Section 4.4.1.
- For every update in the updates file it will be checked if it is a state-change, an update or something else:
  - If the update is a state-change and the previous state was an established connection, all valleys created by that neighbour will be withdrawn from service.
  - If the update is of type update, the relations file will be used to parse the AS path using the algorithm described in Section 3.1. If a valley is detected it will be added to the database.

  - On top of that the update will also be used to detect if it withdraws a currently actively used valley from service using the algorithm described in Section 3.2.

- The next updates file to be parsed will be selected using the scheduler algorithm as described in Section 4.8 and will then be parsed in the same way.

When all files currently available on the server have been parsed the application will start waiting regular intervals for new updates data to arrive on the server and will parse them when they come available.

4.2 Choice for Relation Data

As the relationships between ASes are usually under a non-disclosure policy and we need to know these relations to detect the valleys in the public data, an inferred AS relationships dataset is required. As mentioned in Chapter 2 there have been multiple approaches at inferring relations between ASes, such as Gao [9], Luckie et al. [20] and Oliveira, Willinger, Zhang, et al. [32]. The datasets from the last two are both publicly available for use. Therefore these datasets have been compared to find out which dataset would be best suited for this project. This is done both based on the theory behind them as well as on differences in stability of the actual result.

4.2.1 Differences based on theory

Both CAIDA and UCLA share a lot of similarities in the inference method used; Both make use of publicly available BGP data resources, both make use of the fact that there is a clique at the top of ASes that have no provider themselves and both use this clique to infer customer-to-provider relations for non-clique members.

However, there are also a lot of differences between the two relation inference methods. While UCLA predefines the clique at the top of the topology by manually providing the AS numbers of Tier-1 ASes, CAIDA's approach uses an inference algorithm to automatically establish the set of ASes belonging to the clique. While the UCLA methodology is not prone to errors in the inference of the clique, it does make the assumption that the defined Tier-1 ASes is in fact the same as the top clique, while the CAIDA article argues that "Tier-1 status is a financial circumstance, reflecting lack of settlement payments" and that they "focus on identifying transit-free rather than Tier-1 ASes".

Another difference is that where UCLA explicitly makes inference choices based on the assumption that AS paths should be valley free, CAIDA notes that assuming an AS path is valley-free is not always valid and that their algorithm therefore does not make that assumption. As our application attempts to detect AS paths that violate the valley free property, using an algorithm that relies on the assumption that those paths do not exist would not be very effective.

Also 34.6% of the relations inferred by CAIDA have been validated which have shown that 99.6% of the customer-to-provider relations and 98.7% of the peer-to-peer relations were accurate. According to their article this was "the largest source of validation data for AS-relationship inferences to date". The document containing the algorithm used by UCLA does not mention validation of the data. This makes it hard to assess the correctness of the UCLA inferences.

4.2.2 Relations files compared

Relations between ASes can change over time. ASes can make new peering agreements, or acquire or be acquired by other organizations to become part of a bigger AS which may alter relations between the other ASes. It is therefore to be expected that the relations inferred may change slightly over various months, while most of it should remain stable.

We have compared for the UCLA and CAIDA data what the differences are between the two sources for the same month, as well as what the differences are between the data from the same source, but between each month and the month before it.

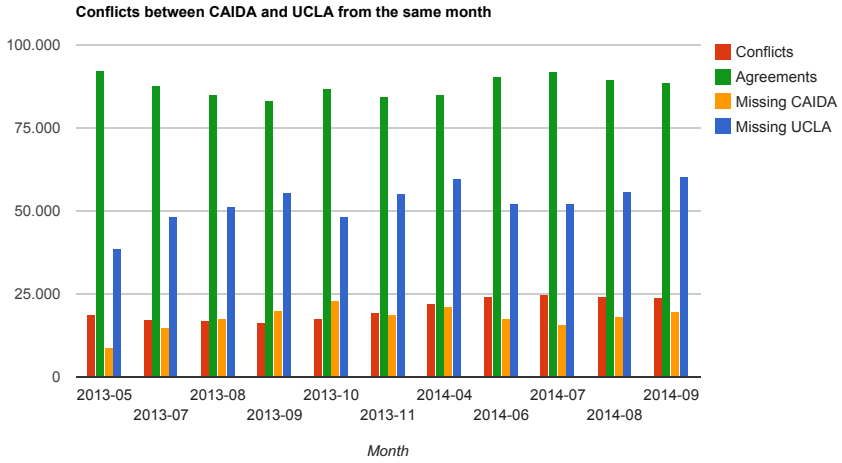
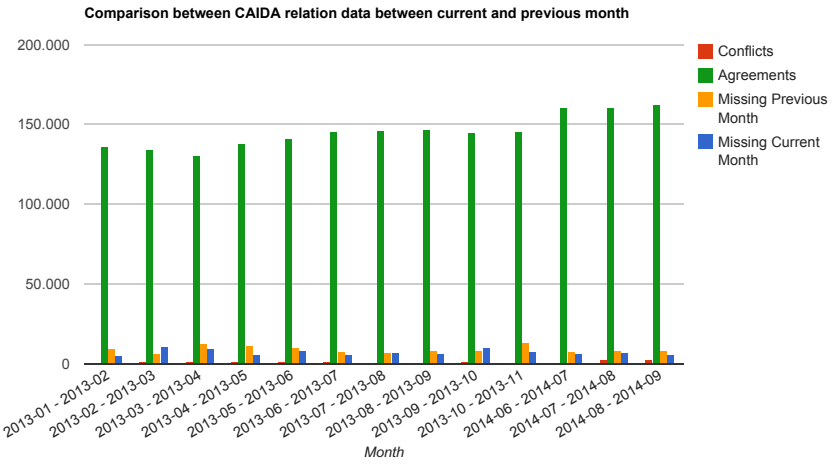


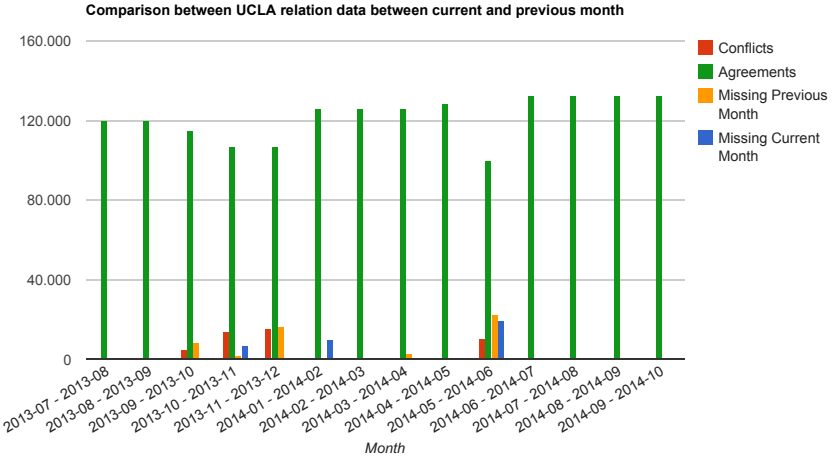
Figure 4.1: Relations for UCLA and CAIDA compared for the same months

Figure 4.1 shows for the months for which both sources had relations data, how many of those relations where equal in both sources (agreements), which ones had a different relation in one source than the other (conflicts) and which ones are unknown in one of the two files (missing).

It can be seen that although the large majority of relations are agreed upon between CAIDA and UCLA, there is also a significant amount of ASes between which CAIDA has inferred a different relation than UCLA. On top of that there are also a few relations that have been inferred by UCLA, but not by CAIDA, but the amount of relations inferred by CAIDA, but not by UCLA is larger. Although apparently CAIDA has inferred more relationships than UCLA, it is not certain if these relations are correct, so a qualitative judgement can not be applied. For the conflicts however, it is certain that either UCLA or CAIDA has inferred the relation incorrectly, so these are certain reason for concern. However, again it can not be determined which one of the inferences is correct.



(a) CAIDA



(b) UCLA

Figure 4.2: Comparison between relation data between current month and previous month

Figure 4.2 shows how much of the relation data changes each subsequent month for the same source.

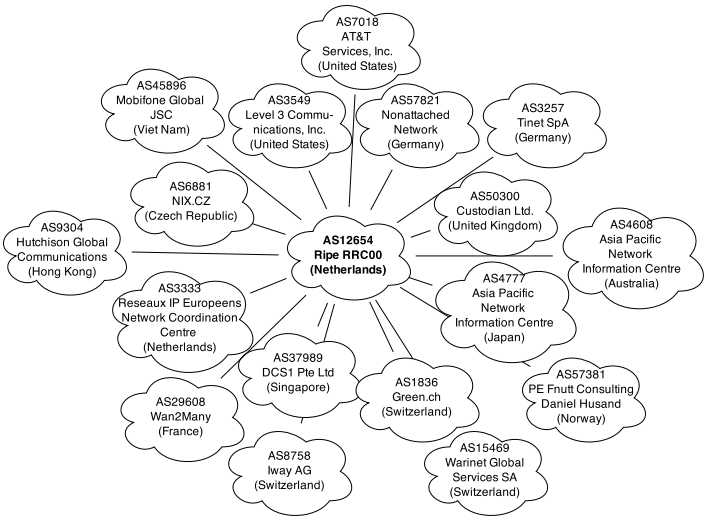
The CAIDA graph shows a minor amount of changed relations each month and a few relations that have been newly inferred or are removed from a subsequent month. The vast majority of relations however remains consistent between months, which is to be expected.

The UCLA graph however, has various months in which not a single inferred relation has changed, but when there are changed relationships, the amount of conflicts between the previous months are much higher compared to CAIDA. It is highly unlikely that not a single relationship is changed for several months, while a significant proportion of the relationships changes

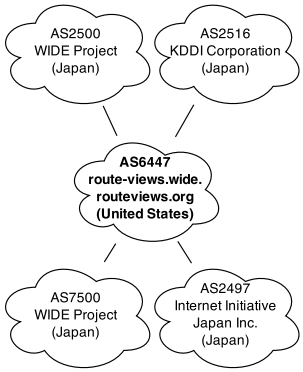
the next months. It is also found that some of the files with 100% agreement rate files are exactly the same in content, while others have only a variation in the relations inferred to be unknown (a inference type that is left out in the CAIDA data).

The relations inferred by CAIDA, although they vary a little every month seem therefore more stable than the relations inferred by UCLA, who either vary nothing at all or a great deal between subsequent months.

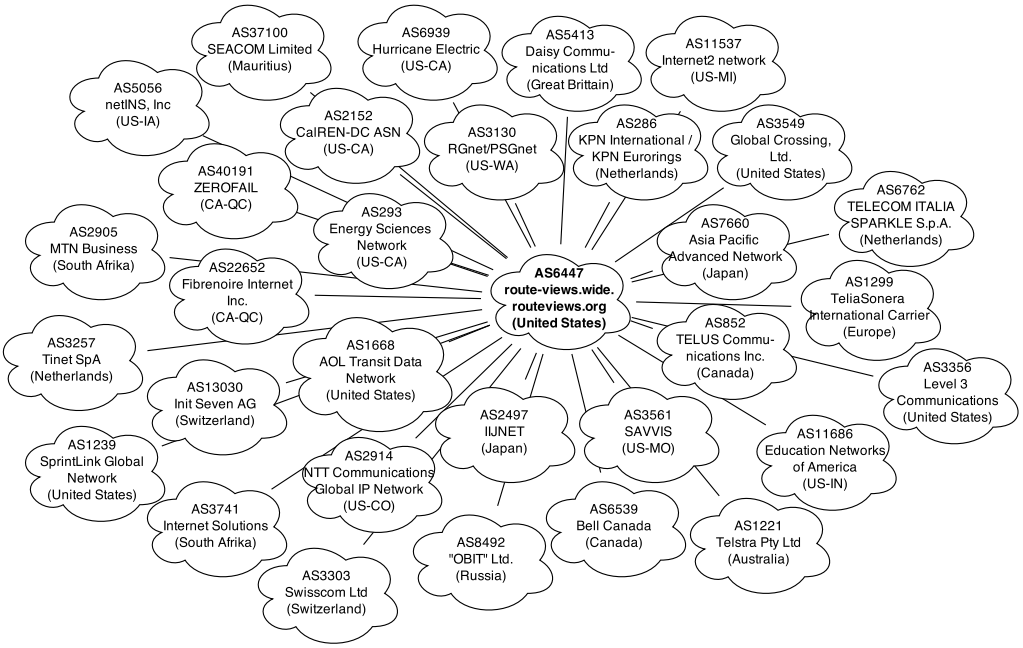
4.3 BGP Traffic Dumps



(a) RIPE RRC00 (located in Europe)



(b) RouteViews WIDE (located in Japan)



(c) RouteViews Oregon (located in the United States)

Figure 4.3: Route collectors and their peers

Besides choosing the relations between ASes, choices should also be made on which data to find valleys in. To maximize the amount of valleys the program will be able to detect the traffic dumps from various route collectors, receiving

updates originating from different locations in the world have been used. For this project we use BGP updates from 3 different sources: from RIPE we use the remote route collector 00 and from RouteViews we use the route collectors Oregon and WIDE.

RIPE RRC00 is a multi-hop route collector which receives updates from various peer ASes. The collector itself is located in the Netherlands and most of its peers come from Europe. However, it also has many big peers coming from other parts of the world such as the Tier-1 ASes Level 3 Communications, inc. and AT&T from the United States, as well as two ASes from Asia Pacific Network Information Centre (APNIC) located in Australia and Japan.

In order to receive more updates from the United States, which we may not see from the RIPE remote route collector due to peering relations in the US, we also added a multi-hop route collector located within the United States. RouteViews Oregon logs the updates received at the Internet Exchange at the University of Oregon. The majority of peers from which the collector receives routes are located in the United States, but there are also some ASes from other parts of the world. Both RIPE RRC00 and RouteViews Oregon have an AS from Level-3 Communications, but not with the same AS number (respectively AS3549 and AS3356).

To receive updates from eastern countries we added a route collector from Japan. RouteViews WIDE is a non-multi-hop route collector which collects only routes received by ASes from the Widely Integrated Distributed Environment project. However, since those WIDE ASes all have other peers from which they receive routes, the routes the WIDE ASes receive will contain various routes not seen by the other route collectors.

4.4 Tools

4.4.1 LibBGPDump

There are several applications / libraries available to parse MRT files. The RIS Raw Data website[38] suggests to use libbgpdump[2] (C) or PyBGPDump[30] (Python), but a Perl implementation [37] exists as well. Since our project is written in Python, PyBGPDump seemed to be the most logical choice. However, this program was unable to parse any of the recent MRT dump files. The cause of this problem seems to be that the library is unable to support the encoding of 32 bit AS numbers as defined in RFC6793[40], which is the common way AS numbers are represented nowadays. After a patch [31] and some manual adjustments the PyBGPDump can work on more

recent files in Python 2.7. As PyBGPdump did not work without the user needing to apply unofficial patches it was first decided to use libbgpdump instead. However libbgpdump also has several disadvantages. First of all it is not native Python and we thus need to execute the application, grab the output and reinterpret it again which results in lesser performance. On top of that since it is a C file it needs to be compiled differently depending on the OS it runs on which would make the whole project less portable. Also when parsing some dump files the program would raise a segmentation fault during execution, which could indicate a potential security issue in the application [3]. Because of the crashes the code also needs to handle more exceptional cases and will miss the possible updates that where within the files that made the program crash, making the end results less reliable. In the end it was decided to take a pure Python approach for the project. In order to remain support for Python 3.x a custom BGP MRT parser module was built, instead of using the patched PyBGPDump library.

4.4.2 MySQL

We use MySQL to store the valleys found in a database. Originally other database types were used (see Appendix A.5), but because of issues with the scalability MySQL is chosen. However, most of the reasons that MySQL was chosen are irrelevant for the current implementation of the application. The original reason to use MySQL in favour of other databases was the support for spatial indexes. For querying the total amount of open valleys at a specific point of time t, we need to find valleys in the database that have a time lower than t, but an end-time higher than t. However, regular binary tree indexes will create a sorting of either time or end-time, but not both. Using spatial indexes solved this scalability issue, but significantly slowed down the procedure of inserting valleys, making the main program to slow. We decided that instead of creating statistics after collecting a lot of data, statistics would be generated on daily bases and the results of multiple days would be combined to calculate the monthly statistics. This way smaller parts of the database are used and when the statistics for a month need to be calculated, only the already present results need to be added up.

4.4.3 Python

The application is written in the Python language. Python comes pre-installed with many Linux distributions, which makes the application easy to install for those systems. The language is also quite easy to learn, which makes it easier for other people to make adjustments to the code when the

application becomes open source. Since Python 2.x is currently the most commonly pre-installed version of Python in Linux distributions, but Python 3.x is expected to take this place somewhere in the future, it was decided to support both versions of Python.

4.5 Building the custom relations file

To find valleys the application needs to know the relations between the different ASes. Instead of using a publicly available AS relation data file as is, we decided to modify one to add awareness of siblings relation to one that originally only defines provider-to-customer and peer-to-peer edges. On top of that, the data source we use [39] does not provide relation data for some of the months we cover. Therefore a method has been created to build a custom relation data file for every month for when that month is not available and to add siblings information to the months of which data is available.

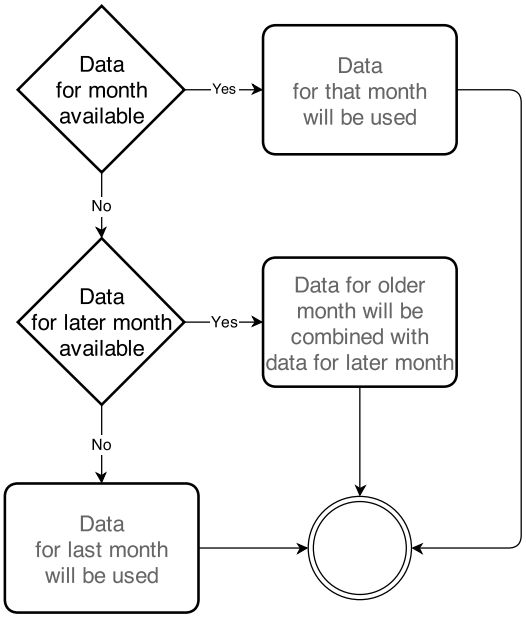


Figure 4.4: Diagram displaying how the relation/siblings data file used is chosen

Figure 4.4 displays how the application chooses which relation data / siblings data file it will use. The same steps will be checked for both relation and siblings data. However, the combining procedure is different.

For the relation data, a combined data set consists of only the relations that are inferred in both datasets to the same relationship. So every relation

that was not inferred in either of the datasets and every relation that has changed between any of the datasets will be removed. This should reduce the amount of false positives. If a relation has changed somewhere within the missing months, it is not possible to determine when exactly this change occurred. Since the changed relation is removed from the dataset, the relation will no longer cause valleys to be detected.

For the siblings data, a combined data set consists of all siblings that appear in either of the two datasets. As the siblings data is used to reduce the amount of false positives by labelling detected valleys that are not really valleys when the siblings data is used, a combined dataset would label more valleys to be non-real valleys and thus further reduce the amount of false positives in the real valleys data. Since we do not know when the ASes that were not siblings in one dataset became siblings in the other dataset we therefore join all siblings from both datasets that surround the month that will be parsed.

4.6 Analysis queries

4.7 Analysis queries

After parsing all the updates for a certain day for a specific source, queries are performed to fill the statistics tables in the database and their results are stored in the appropriate tables. As seen in 3.3 there are 5 different stats tables which will be filled using the results of different queries. Every one of those tables will get filled with data per source and per IP version.

4.7.1 Violation Types

For the *stats_violation_types* table, the following query will be performed for every *violation_type*:

```
SELECT COUNT(*)
FROM valley WHERE
violation_type = %s AND
timestamp BETWEEN %s AND %s AND
source_id = %s AND
ipv6 = %s;
```

With as the first argument the *violation_type* that is currently queried, the second and third arguments the first and last millisecond of the day for which statistics are being made and the last two arguments the source and whether

or not IPv6 statistics are being made. This query will simple count all valleys found that had the specific violation type on a single day for the specified source/IP version combination.

4.7.2 Valley Duration

For the *stats_valley_duration* table a similar query is performed:

```
SELECT COUNT(*)
FROM valley WHERE
duration BETWEEN %s AND %s AND
violation_type IN (1, 2, 4, 8) AND
timestamp BETWEEN %s AND %s AND
source_id = %s AND
ipv6 = %s;
```

The query only counts violation types of 1, 2, 4 and 8 which are real relation types P2P-C2P, P2P-P2P, P2C-C2P and P2C-P2P and not the other relations which have been inferred as being siblings. In the query the duration should be between two values determined by the duration index. The duration ranges we check increase in range exponentially and follow the pattern $<= 2$, $3 - 4$, $5 - 8$, ... $>= 524289$. The last one also includes duration values that have not been set yet and thus instead of the normal duration BETWEEN part it will have:

```
duration >= 524289 OR duration IS NULL
```

Since the duration may be NULL for valleys that will eventually be closed earlier than after 6 days and thus are incorrectly inferred as having a duration of over 6 days, this query will be performed after 7 days as well and will overwrite the previous values.

4.7.3 Open Valleys

For the *stats_open_valleys* table the following query is performed:

```
SELECT COUNT(*)
FROM valley WHERE
timestamp >= %s AND
end_timestamp <= %s AND
violation_type IN (1, 2, 4, 8) AND
source_id = %s AND
ipv6 = %s;
```

Here both the values for *timestamp* and *end_timestamp* will be set to the day being looked at at 08:00 (arbitrary value). This will count all open "real" valleys that have been announced before the specific time, while they were not withdrawn before that time.

4.7.4 AS Triplets Top

The queries for the *stats_as_triplets_top* table work somewhat different than the previous ones. Here we group the data per triplet (*leak_as*, *leaked_from* and *leaked_to*), select only the valleys with a duration of more than a minute and order by the amount of occurrences. Since we do not have infinite space, we limit the amount of results to contain only the 100 most occurring triplets that meet the constraints.

```
SELECT leak_as, leaked_from, leaked_to,
COUNT(*) as amount
FROM valley WHERE
duration >= 60 AND
violation_type IN (1, 2, 4, 8) AND
source_id = %s AND
ipv6 = %s
ORDER BY amount DESC
GROUP BY leak_as, leaked_from, leaked_to
LIMIT 100;
```

4.7.5 Country Leak Top

The queries for the *stats_country_leak_top* table work similar to the ones from AS Triplets Top, but with instead of the fields from the leak triplet, the country_code will be grouped by. Since the country_code is not part of the valley table, the *as_location* table needs to be joined to it first.

```
SELECT country_code, COUNT(*) as amount
FROM valley
JOIN as_location ON valley.leak_as = as_location.asn
WHERE duration >= 60 AND
violation_type IN (1, 2, 4, 8) AND
source_id = %s AND
ipv6 = %s
ORDER BY amount DESC
GROUP BY country_code
```

```
LIMIT 100;
```

Before this query is performed it is made sure that all leaking ASes of that day have a Country Code added to the *as_location* database to prevent certain leaking ASes to have no matching Country Code.

4.8 The scheduler

The system that decides which file containing updates should be parsed next and when is a complex system because it tries to meet many requirements. The requirements that have been considered while making the scheduler are as follow:

- There should never be any updates file skipped, because when any update is not parsed, a valley or the closure of a valley may be missed. Therefore, when the application reboots, the last valley parsed will be re-parsed again to see if all updates were already handled. After that always a new updates file will be selected which directly follows the previous one found.
- It should support different sources for updates files and it should regularly switch from which source it returns a updates file to parse, so no source gets behind schedule.
- The file name of the next file containing updates should be determined automatically. Usually the file times are predictable (e.g. 5 or 15 minutes between them), however it should also correctly determine the file names that stray of from the default pattern.
- When files are already downloaded before, it should use those files and don't re-download them.
- When files are not downloaded yet, it should download them from the source, but not too much at the same time to prevent the server from being overloaded with requests.
- When a updates file cannot be downloaded, the reason should be traced and correct actions should be performed:
  - When the server does not respond, an appropriate delay should be used before retrying. This delay should increase with every failed attempt and decrease when a file is successfully downloaded.

– When the server does respond, but the file is unavailable, while other files uploaded later are available, the file name will not have followed the regular pattern and the alternative file name should be found.
– When the server does response, but the file is unavailable, and there are no later files available as well, the application should wait until the file is uploaded. After this event, all subsequent files will need to be uploaded too, so the system should wait before attempting to download subsequent files as well. This delay is usually the same as the differences between two subsequent file times, but occasionally also differs a lot which should also be handled correctly.

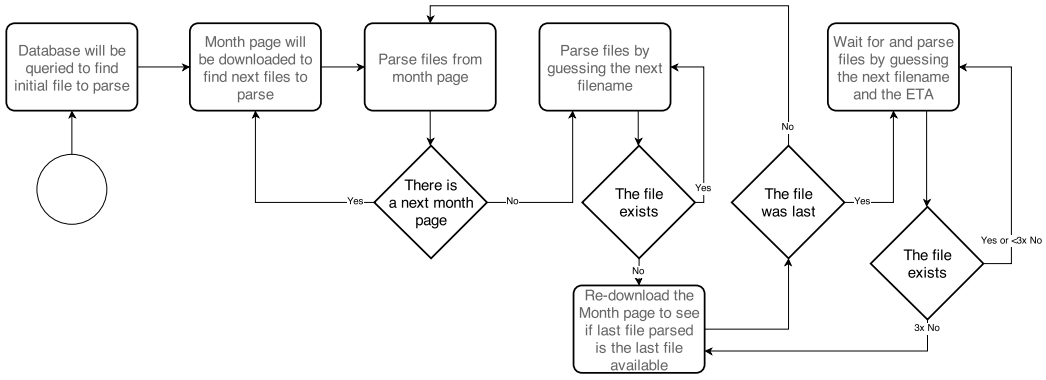


Figure 4.5: Diagram displaying how the updates time to parse is chosen

Figure 4.5 shows how the file to be parsed next will be selected. The same procedure is followed for every source and the one with either the lowest file time that will be downloaded next, or the source for which the ETA until the next file is available will be chosen next.

The first file to be parsed will be the file containing the last valley available in the database. This valley will be re-parsed to check if all valleys available in that file had been added to the database the last time the program was executed.

After the first file has been parsed, the index page for the same month will be downloaded or loaded from cache and the files found after the first file will be parsed in the correct order. If the month contains all updates files it will ever contain, the next month will be downloaded as well and all updates from that month will be parsed, which will be repeated until the last update from the current month is parsed.

When the last updates file available on the current month page, at the time that that page was downloaded, has been parsed the consecutive filenames will be "guessed" by adding the usual difference between the last file times (usually 5 or 15 minutes) to the last file parsed. While the filename is correctly guessed (and thus exists), the next filename guessing will continue. When the filename does not exist however, the month page index will be downloaded once more to see if the last file parsed was indeed the last one currently available.

When this is the case, the next file will be guessed again, but it will be downloaded only after a certain interval (the same as the time difference), as the server will only upload new files every so many minutes. When a guessed filetime is wrong multiple times in a row, the month page will be re-downloaded again to prevent the program lock up due to misnamed files.

This approach will not make too many more requests then strictly necessary, while still downloading the latest updates files close to as soon as possible. There are more delays not described in this figure to handle server timeouts, while parsing files according to the month page for example, that further reduce the stress put on the servers, but those have been left undiscussed to reduce the complexity a little.

4.9 Web based interface

Apart from the main statistics presented in this paper, we also created a web based interface for users to view the data derived by the project's application. The web interface contains two parts: the query part and the statistics part. The query part allows users to find in the database route leaks that fulfil certain conditions (e.g. is a certain AS number causing a route leak). The statistics part allows the user to view certain statistics such as the ones presented in this document. In this section it will be described how the web based interface works and how it gets the information from the database.

4.9.1 Query server

Apart from the main application, another application will be running on the server which is the Query Server. This Python script waits for incoming HTTP requests and responds in JSON format. A request will be in the format of GET /valleys/{key1}/{value1}/{value2}/{key2}/{value3}/...etc. Where if multiple values are given for a single key, it will be either interpreted as a OR clause or a BETWEEN clause depending on the key. The response will be a JSON document containing a list of the found route leak updates with

their information. To prevent SQL injections from occurring, none of the keys or values will be directly used in the SQL query. Numbers will be pre-casted to integers, IPs and prefixes will be pre-converted to their binary form and the source will only allow values that are defined in the configuration file. To reduce the impact of later adjustments to the source that possibly introduce SQL injections from occurring, the Query Server also runs on a different MySQL user than the main program, which only has read access to the database.

4.9.2 Statistics

As written in Chapter 3, the main application also frequently generates statistics which it records in the statistics database. The Query Server also accepts requests in the format "GET /statistics/day_between/{day_from}/{day_to}". The response will again be a JSON document with the values needed to produce certain graphs. To produce the graphs the website will use Google Charts [14]. Google Charts was chosen over alternative solutions because it provides a single method to visualize the various charts (e.g. pie chart, bar chart, but also geo chart) we wanted to use. On top of that it was relatively simple to use while it still has various advanced options to make it customizable enough for our needs.

4.10 Validation

4.10.1 Known Limitations

There are several limitations with the approach we are using. In this section we will be discussing what limitations are known and what is done to reduce the effects of it.

Relying on AS relations

First of all, the project heavily depends on the relationships between neighbouring ASes. Since the actual relations between ASes are not known, inferred data sets are used. To minimize the amount of false positives we have chosen the most reliable AS relationships data set we could find (as motivated in Section 4.2) and combined them with siblings inferences to reduce the amount even further (as described in Section 4.5). However, it is still possible that some AS relations have been inferred incorrectly and this will result in some valley detections where there are none.

Apart from the possibly wrongly inferred relations, the relational data also contains only one relation for every AS pair. As we have seen in Section 2.2.2, there are ASes that have different relations based on the IP protocol version being used. Since we do not have separate relations data for the different IP versions, we will not be able to distinguish an alternative relation for a given protocol from a violation in the policy. Therefore if the relation between two ASes is different for IPv4 and IPv6 and the relation data contains only the relation inferred for IPv4, updates for IPv6 will likely be incorrectly parsed as being a valley.

Spurious routes

According to Luckie [19] the AS path attributes in the public BGP data cannot always be relied upon. Since there is currently no mechanism in BGP to prevent the AS path being manipulated differently from the specification (prepend your AS number at each external forward) any AS in the path can choose to manipulate the AS path.

If, for example, AS1 does not want AS2 to take the path he received from AS3, AS1 may prepend AS2's number as well before announcing it to its neighbours. This way when AS2 receives this route later on its loop prevention algorithm will prevent this path from being used. However, this will also affect the AS path received at a vantage point as this will also receive the route which has been tampered with. Therefore we will interpret it as that AS3 has send the path to AS2, who send the path to AS1, even though the path was send from AS3 to AS1 and AS2 was never in between this path.

Luckie has defined a method to identify some of the spurious routes by utilizing and combining data from various vantage points. Implementing this method however would take too much work for the scope of this project and would heavily increase the computational complexity of the project's application. The expected amount of spurious routes is not deemed significant enough to have a noticeable effect on the project's statistics. Furthermore BGPsec[18] will eventually secure the AS path attribute, making it impossible for ASes to create such spurious routes between ASes which have BGPsec implemented.

Chapter 5

Results & Analysis

We decided to focus our analysis efforts on the following aspects:

- *Violation Types*; This shows whether or not certain combinations of relations causing a valley are more frequently occurring than others. On top of that, we also left in this chart the valleys that were actually caused by siblings relations, so the effect of using siblings data can be looked upon as well.
- *Valley Durations*; This graph provides means to see how long it takes until a route becomes unavailable to use. This way it can be seen if route leaks are quickly being fixed or that they keep active over a longer time span.
- *Open valleys per day*; This graph shows the total amount of active valleys at a certain point of time every day. This way it can be seen if the amount of route leaks is growing, shrinking or remaining consistent over time.
- *Top 10 triplets*; This chart views the most frequently occurring AS leak triplets (as defined in Section 3.1.2). These triplets have been further investigated to see if we can determine something about whether or not the most frequently occurring valleys found are intentional (documented) or not.
- *IP versions*; According to Giotsas and Zhou [12] valley paths occur relatively more frequent in routes for IPv6 prefixes than they do for IPv4 addresses. To see if we can confirm or disprove this statement, statistics are also created specifically for the different protocols.

- *Regionality*; Different countries have different policies on how their inhabitants can reach certain parts of the web. Occasionally to block certain websites BGP is used [5]. It may be interesting to see if we can detect whether or not certain regions of the world that have a stricter control over their population also are responsible for a larger amount of route leaks.

5.1 General Results

(a) IPv4 counts

Source	Updates	Valleys	%
RIPE	443034456	18778363	4.24%
Oregon	1043108312	32179313	3.08%
WIDE	24441242	1126113	4.61%
All	1510584010	52083789	3.45%

(b) IPv6 counts

Source	Updates	Valleys	%
RIPE	40987405	9322831	22.75%
Oregon	38272251	4040745	10.56%
WIDE	3145674	762861	24.25%
All	82405330	14126437	17.14%

Figure 5.1: Total number of updates and valleys per source and IP version

Figure 5.1 shows the number of updates counted in all the updates files in the range of data we examined, along with the amount of valleys that were detected in those updates for the sources used for IPv4 and IPv6. The number of valleys displayed here still include the valleys that were later filtered out as they were determined to contain a siblings relation.

The results show that the amount of valleys detected in IPv6 relative to the amount of updates is much larger than in IPv4. In Section 2.2.2 there were two possible explanations for the difference in results for IPv4 and IPv6 data. Giotsas and Zhou [12] mentioned that the IPv6 protocol is not yet implemented by enough ASes to allow for global reachability without creating valleys and that thus valley paths will be accepted as well.

However, they also noticed that there are ASes which have different relationships based on the protocol used. Since we only have one relation inferred for every AS combination and IPv4 has significantly more updates than IPv6, it is likely that the inferred relation applies to IPv4. When the IPv4 relation is then used to find valleys for IPv6 prefixes, this will result in false positives when the IPv4 and IPv6 relations are not the same.

5.2 Violation Types

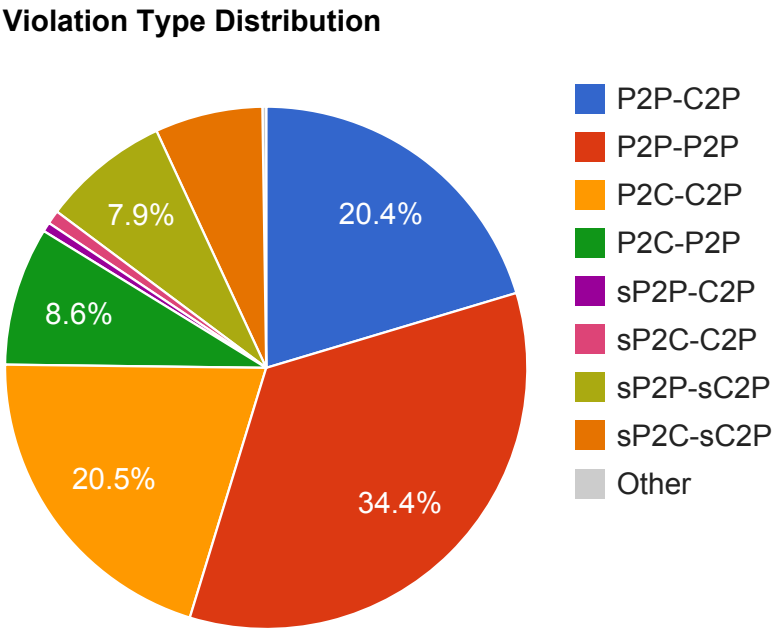
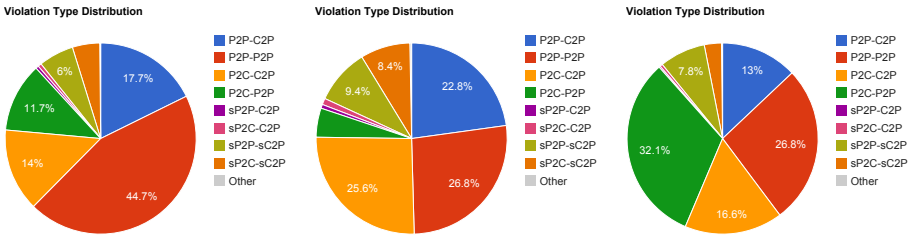


Figure 5.2: Violation Types (of valleys found between 2014-04-12 and 2014-06-12)

Figure 5.2 shows the distribution of violation types of all valleys found between 2014-04-12 and 2014-06-12 in all of the BGP traffic sources for all IP versions. Since we added the siblings inference later on in the project, we also divided the route leaks that turned out to be siblings relations from the real route leaks. Here we see that about 16% of the valley free violations were later found to be caused by siblings relations inferred as different relations. In the other graphs only "real" valleys will be handled.

From the "real" violation types, the P2P-P2P violation appears to be the most prominent. The frequency of this type of violations can be caused by various reasons: As discussed earlier in Section 2.2.2 ASes can perform indirect peering at an IXP and sometimes these IXP ASes are not removed from the AS path. It may also be possible that ASes have complex policies that require them to share peer routes with other peers. In this case one AS will provide transit between those two peers' client routes and may be compensated for doing so. However, this is purely an hypothetical explanation as we have no confirmation of the existence of such policies between ASes.



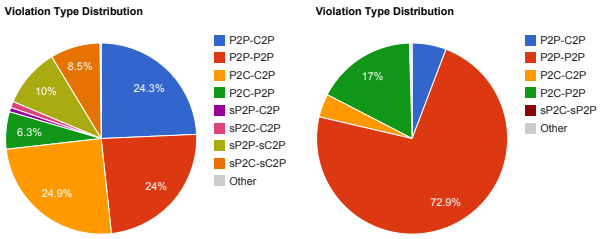
(a) RIPE (b) Oregon (c) WIDE

Figure 5.3: Violation types per source

In Figure 5.3 charts have been made for the same period, but now the distribution is charted per source for BGP data. The RIPE chart (Figure 5.3a) has a lot of similarities to the overall chart (Figure 5.2), but there are some differences. The most noticeable change is that the P2P-P2P violation type is even more dominant (44.7% v.s. 34.4%).

In the Oregon chart, the P2P-P2P violation type still has the biggest share (26.8%), but the P2P-C2P and P2C-C2P violation are almost equally distributed (22.8% and 25.6% respectively). This seems to indicate that the valley free violations seen by one route collector can differ significantly to the ones seen by a route collector located elsewhere.

In the WIDE chart, the P2C-P2P, which is the smallest "real" violation type in the other charts, is the most prominent violation type (32.1%). A likely cause of this effect is that the direct peers of RouteViews WIDE (Figure 4.3b) are part of the WIDE project[35], which is a research institution. As previously discussed in Section 2.2.2 research and educational ASes are responsible for a significant amount of prevalent valley-free violating routes that are propagated. As will be seen in Section 5.5 this indeed seems to be the case as the most frequently occurring route leaks involve a WIDE AS and one or more other research/educational ASes. As most of the other research/educational ASes are inferred to have a peering relation with the WIDE project, and the routes they share are mostly inferred as being provider routes, these valley-free violations are overrepresented in this chart.



(a) IPv4 (b) IPv6

Figure 5.4: Violation types per IP version

Figure 5.4 shows the charts for distribution of violation types for IPv4 and IPv6 prefixes. The IPv4 chart mostly resembles the original Figure 5.2. However it can be seen that the P2P-P2P violation type has been reduced by about 10%, which is now distributed over certain other violation types.

Figure 5.4b explains the difference in P2P-P2P violations percentages, as in IPv6 this violation type is massively overrepresented. The huge amount of P2P-P2P violations for IPv6 prefixes is mainly caused by a single AS which will be seen in Section 5.5. Since this AS causes more than 70% of all valleys found in IPv6 prefix announcements and the majority of its detected valley free violation are of type P2P-P2P, this should explain the P2P-P2P violation type being significantly larger in the IPv6 chart than it was in the other charts.

5.3 Valley Duration

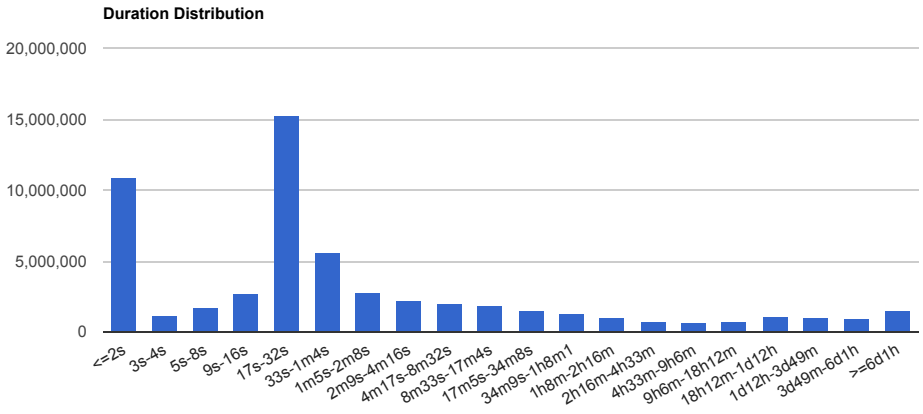


Figure 5.5: Valley Durations

Figure 5.5 shows the distribution of the durations of the valleys found. The duration is the time between the announcement of a valley and the withdrawal of that announcement from service. The valleys used are within the same date-range as the ones in Section 5.2, but only the "real" valleys are used, not the ones that were actually siblings.

We see two very noticeable peaks at the less than 2 seconds (around 10 million) and the between 17 and 32 seconds (around 15 million) ranges. These announcements are very likely part of route convergence and not intended as a final route and thus withdrawn immediately. As in most routers the default MRAI timer (minimum route advertisement interval) is 30 seconds, a fixed route cannot be propagated within the first 30 seconds, which explains why there is a peak in the 17 till 32 seconds range. The reason why there is also a peak at the less than 2 second range is that, according to a recent survey [10], a lot of network operators disable this MRAI timer allowing the incorrectly announced routes to be withdrawn from service immediately.

Seeing that most valleys are very short-lived it seems as though most of them are not intentionally announced, because the amount of time that route will be used for routing purposes is negligible. However it can also be seen that there are quite a few route leaks left that have a duration of hours or even days. When counting all route leaks that take longer than 1 hour and 8 minutes there are still over 6 million route leaks with these durations. If these routes are not intentional, they do impact the routing for a significant amount of time.

5.4 Valleys over time

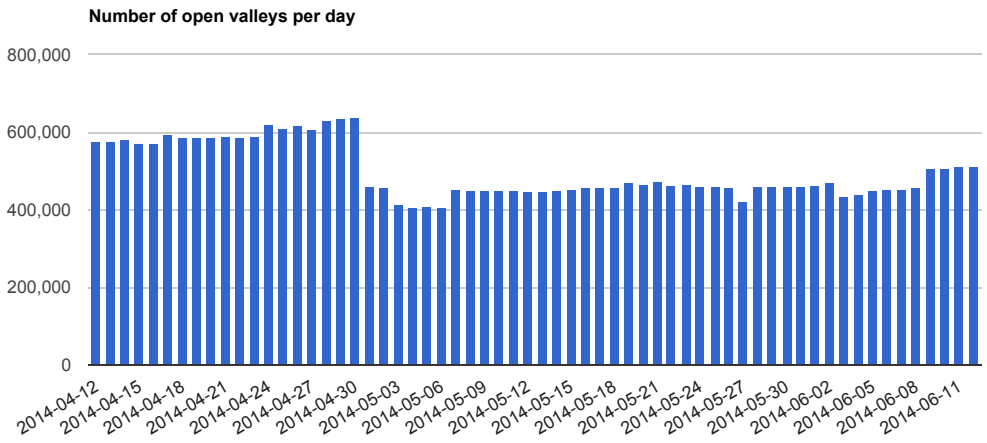


Figure 5.6: Valleys over time

Figure 5.6 shows per day the total amount of valleys that have not been withdrawn at 8:00 for all sources. We see the amount of valleys does not increase quickly and it often drops down again, making it uncertain if there is a growing trend in the total amount of valleys. What can be seen is that there is a big drop down in the amount of valleys at 2014-05-01. Since we use relation data that changes monthly, valleys stored in the database will be closed when the new relation data does no longer infer that valley as having a valley free violation in it. Apparently about 28% of all the routes detected as a valley before 2014-04-30 with the relation data from month 2014-04 are no longer a valley according to the relation data for the month 2014-05. As there is no relation data from CAIDA for this month a combined relation file is created as described in Section 4.5, which can further explain the big drop in total amount of valleys.

This indicates that the freshness of the relation file used has a significant influence on what will be seen as a valley. It also seems to indicate that to accurately find valley free violations one new relation data file per month may not be frequent enough.

5.5 Top 10 triplets occurrences

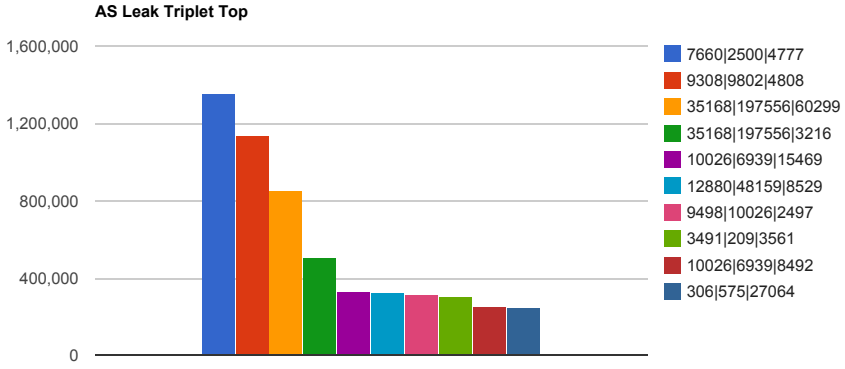
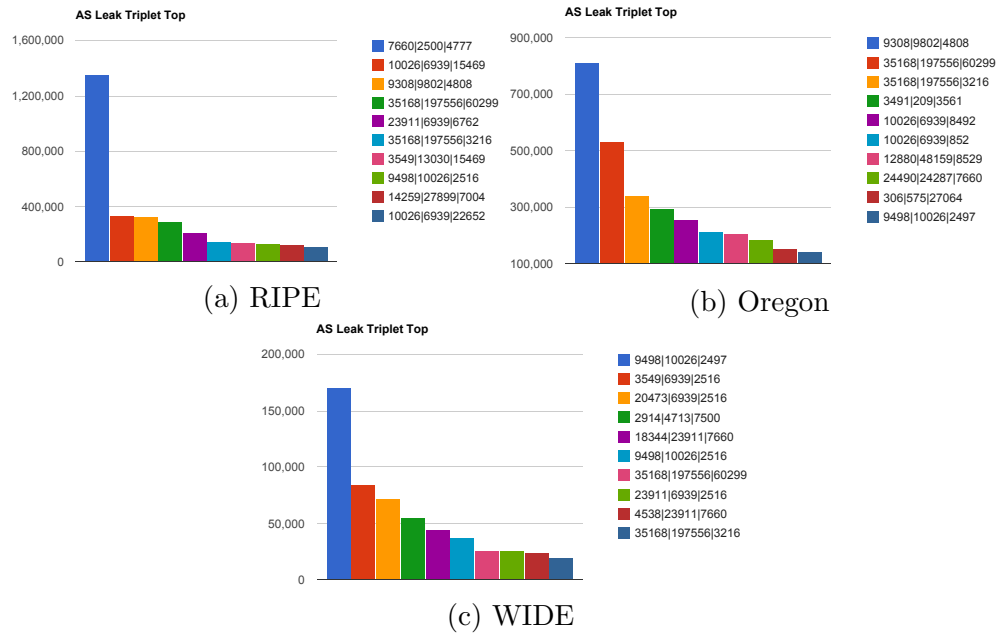


Figure 5.7: Top 10 AS leak triplets

A valley is created by three ASes in a row that have relationships that should not follow each other according to the valley free rule. The three ASes that form these valleys we call the AS leak triplet. In Figure 5.7 the top 10 most frequently re-occurring AS leak triplets have been charted. Because it was seen in Section 5.3 that a majority of the announced valleys have a negligible duration, it was decided to count only the valleys with a duration of 1 minute or higher. The total amount of valleys in the date range used that conform

to this constraint is about 19.5 million. In Figure 5.7 the top 10 AS leak triplets together comprise 5632337 of these valleys. This means that the top 10 AS leak triplets are responsible for creating about 29% of all valleys.



(a) RIPE (b) Oregon

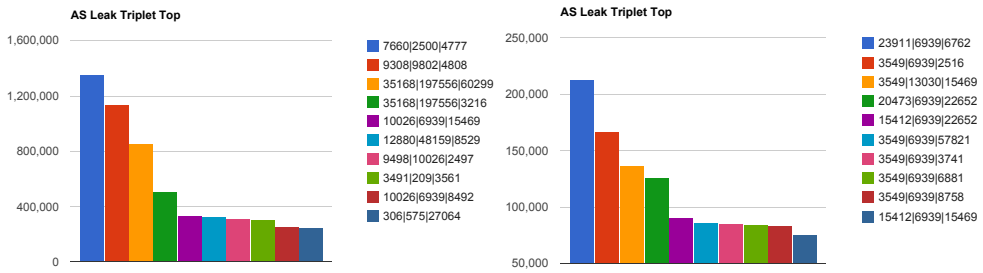
(c) WIDE

Figure 5.8: Top 10 AS leak triplets per source

Figure 5.8 shows the top 10 AS leak triplet top 10 for all of the individual route collectors we analysed. The top leak triplets observed by various route collectors varies greatly. The overall most frequently occurring leak triplet 7660|2500|4777 is found only in the RIPE route collector, and not in any of the other route collectors used. This specific case will be discussed more thoroughly in Subsection 5.5.1.

The second overall most frequently occurring leak triplet 9308|9802|4808 is the most frequently occurring leak triplet in the Oregon data. However, unlike the previous one, this triplet is also found in the RIPE top 10 (position 3) and while it does not show up in the WIDE top 10, the triplet does appear there as well.

The third overall most frequently occurring leak triplet 35168|197556|60299 can be found in top 10 of all three of the route collectors (RIPE: #4, Oregon: #2 and WIDE: #7) In the following subsections, some of the found valleys will be further analysed to find out why they occur and why they either appear at only a few route collectors or at all of them.



(a) IPv4 (b) IPv6

Figure 5.9: Top 10 AS leak triplets per IP version

When comparing the valleys appearing while IPv4 prefixes are announced with the once formed when IPv6 prefixes are announced it seems that there are no AS leak triplets that appear in both lists. Also since the most frequently announced IPv6 leak triplet is announced 212202 times, which is lower than the 10th most frequently announced IPv4 triplet, the top 10 of IPv4 is the same as the overall most frequently announced AS leak triplet top 10. What is noticeable about the IPv6 chart is that AS6939 (Hurricane) appears in all but one of the leak triplets listed in the top 10. This is likely caused by Hurricane having different relationships for IPv4 and IPv6 routes. The Hurricane valleys will be discussed further in Subsection 5.5.4.

5.5.1 The APAN - WIDE - APNIC valley

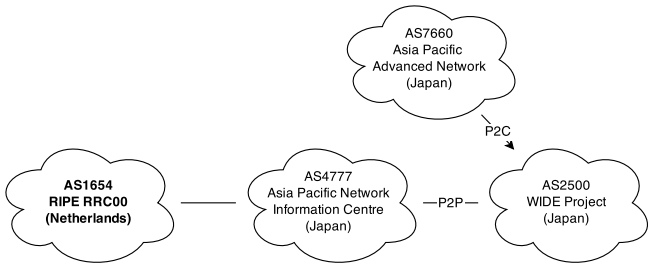


Figure 5.10: Valley APAN - WIDE - APNIC

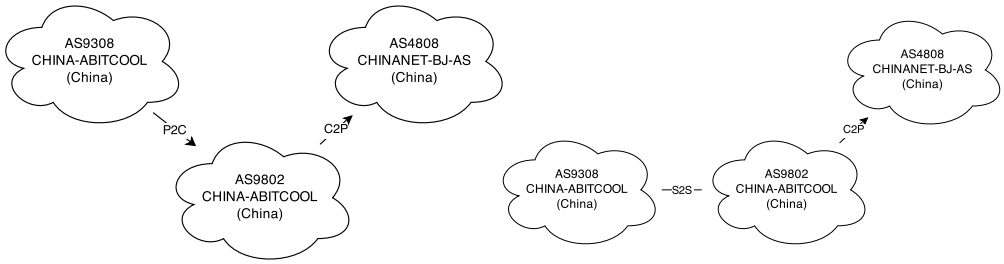
The overall most frequently occurring leak triplet found was 7660|2500|4777. In Figure 5.10 the ASes related to this valley have been drawn.

In this situation AS2500, belonging to the WIDE project, appears to be leaking a provider route, received from AS7660 (APAN) to a peer AS4777 (APNIC). Therefore WIDE seems to provide free transit through APAN for APNIC.

However, all of these ASes are found to be related to research/educational ASes that often do not follow regular relationships such as the once inferred by CAIDA. APAN (the Asia Pacific Advanced Network) is a "backbone network that connects the research and education networks of its member countries/economies to each other and to other research networks around the world"[1]. The WIDE project is involved in "integrating academia and industry into a unique consortium to help researchers with free and unrestrained innovation overcome the traditional boundaries of organizations and utilize new technologies to create a better society and achieve their own self-realization."[35] The Asia Pacific Network Information Centre (APNIC), is the not-for-profit regional Internet registry for the Asia Pacific region.

Because of the nature of these organizations and because WIDE also mentions [42] to be related to both APAN and APNIC it is likely that WIDE is intentionally providing transit for APNIC through APAN. Since APNIC is a direct peer of the RIPE route collector, this valley is observed at this collector, but since this route is only used by APNIC and not announced further, it does not appear at the other route collectors.

5.5.2 The aBitCool valley



(a) Valley according to CAIDA relation data

(b) No valley when AS9308 and AS9802 are siblings

Figure 5.11: The aBitCool valley

The second overall most frequently appearing AS leak triplet was 9308|9802|4808 which is shown in Figure 5.11a. Here AS9802 appears to leak a provider route from AS9308 to its other provider AS4808.

However, both AS9308 and AS9802 share the same AS name "CHINA-ABITCOOL". This seems to indicate that both belong to the same company aBitCool and that they would not have a P2C relation, but a siblings relation. As mentioned in Section 2.2, siblings can share any route and may appear anywhere in the path without causing a valley.

To reduce the amount of valleys detected because of siblings relations we used the CAIDA AS2Org data as mentioned in Section 2.3.1. However, this data had not inferred these ASes as belonging to the same organisation. (AS9308 was assigned organisation ID @aut-9308-APNIC, while AS9802 was assigned organisation ID @family-35609)

The reason why the two ASes were not inferred as being siblings is uncertain. Looking into the WHOIS data, both ASes share many similarities. However, many fields also have the same information with minor alterations, which may confuse the algorithm that CAIDA used.

For example, the admin and technical contact for both ASes is Wei He, but they do not share the same handler ID, thus linking to two different persons. On top of that, the person name for the one person object is Wei He, while in the other person information it is filled in as He Wei, making it hard for automated tools to see them as identical. However, the policy of AS9802 describes that it announces all ASes belonging to AS-21VIANET (21Vianet is an old name of aBitCool), containing AS9308, AS9802 and various other aBitCool siblings to AS4808, indicating that AS9308 and AS9802 indeed are siblings and it is incorrectly inferred as being a valley.

5.5.3 The TNS Plus valleys

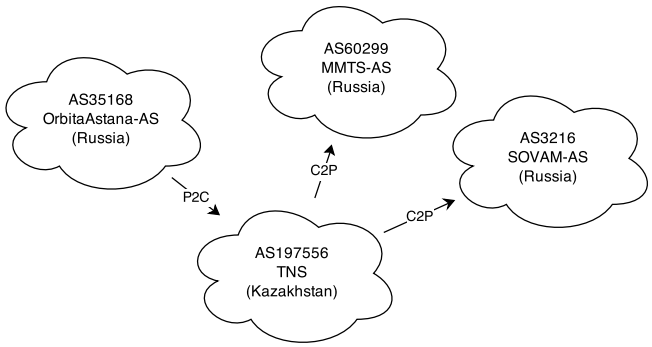


Figure 5.12: Valleys TNS Plus

The third and the fourth overall most reoccurring AS leak triplets are both TNS Plus announcing a route from its provider Orbita Astana to two different providers. However, both providers to which the routes are being announced have defined in their routing specification to accept from TNS Plus, routes listed in the AS-SET_TNSPLUS group. This group contains Orbita Astana and various other ASes that appear before that AS in the AS path. It therefore seems plausible that Orbita Astana is not a provider for TNS Plus,

but that TNS Plus is a provider for Orbita Astana. If this would be the case, TNS Plus would not be leaking a provider route, but just announcing a customer route, which would mean there is no route leak at all.

5.5.4 The Hurricane valleys

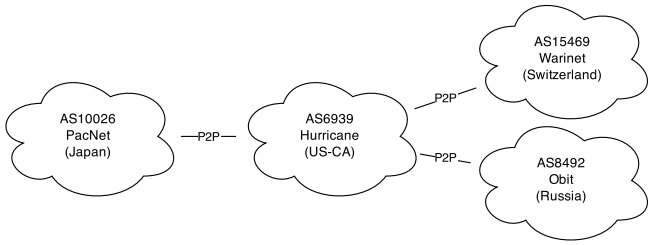


Figure 5.13: Valleys Hurricane (IPv4)

The fifth and ninth most frequently occurring route leaks both involve Hurricane announcing a route from its peer Pacnet to another peer, such as Warinet or Obit. Since neither Pacnet, nor Hurricane, nor Obit provide RPSL data to the WHOIS databases investigated, it is hard to tell what the intended policies are. However Warinet does provide RPSL information which shows that for IPv4 they are indeed peers ("mp-import: afi ipv4.unicast from AS6939 action pref=150; accept AS-HURRICANE"), while for IPv6 Warinet has configured Hurricane as being a provider ("mp-import: afi ipv6.unicast from AS6939 action pref=150; accept ANY").

However, the valley is detected in announcements containing IPv4 prefixes, indicating that if the relation between Pacnet and Hurricane is indeed a peering relation, Hurricane will indeed be providing transit between two peers. However, AS-HURRICANE, which contains all the ASes of which Warinet accepts routes from when received from Hurricane, contains also AS10026 (Pacnet) which seems to indicate that either Pacnet is a client of Hurricane or that it is for another reason intentionally providing transit between those networks.

Feedback from Hurricane indicated that both PacNet and Warinet are clients of Hurricane and not peers. This indicates that the CAIDA relationship the program relies upon seems to incorrectly infer peering relationships for Hurricane with some of its clients which results in incorrect detections of P2P-P2P valleys.

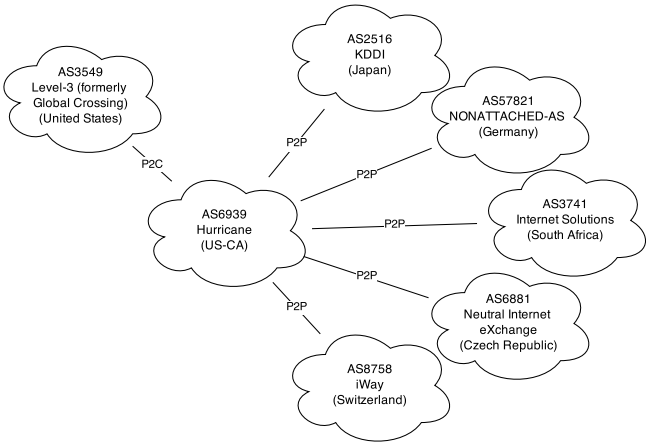


Figure 5.14: Valleys Hurricane (IPv6)

Apart from the appearances in the general top 10 leak triplet occurrences (figure 5.7), Hurricane also is indicated as the 'leaking AS' in most of the entries in the IPv6 chart (figure 5.9b). In most of these occurrences it looks as though Hurricane is leaking a route from its provider AS3549 (Level-3) to one of its peers.

When performing a WHOIS lookup for the as-set *AS-HURRICANE*, which contains a list of all clients for Hurricane, we learn that AS6881 has been defined as a client of Hurricane instead of a peer. Therefore it is likely that this particular case is inferred incorrectly. When performing a WHOIS lookup for the as-set *AS-HURRICANEv6*, which contains a list of all ASes that are a client of Hurricane for IPv6 traffic, we notice that all "peers" according to Figure 5.14 are actually clients for IPv6 traffic. This indicates that Hurricane has different policies for traffic over different IP versions and thus that using a relation inference which only outputs relations based on one protocol will cause a lot of false positives to be found when used to find valleys in the other protocol.

5.6 Leaks per country

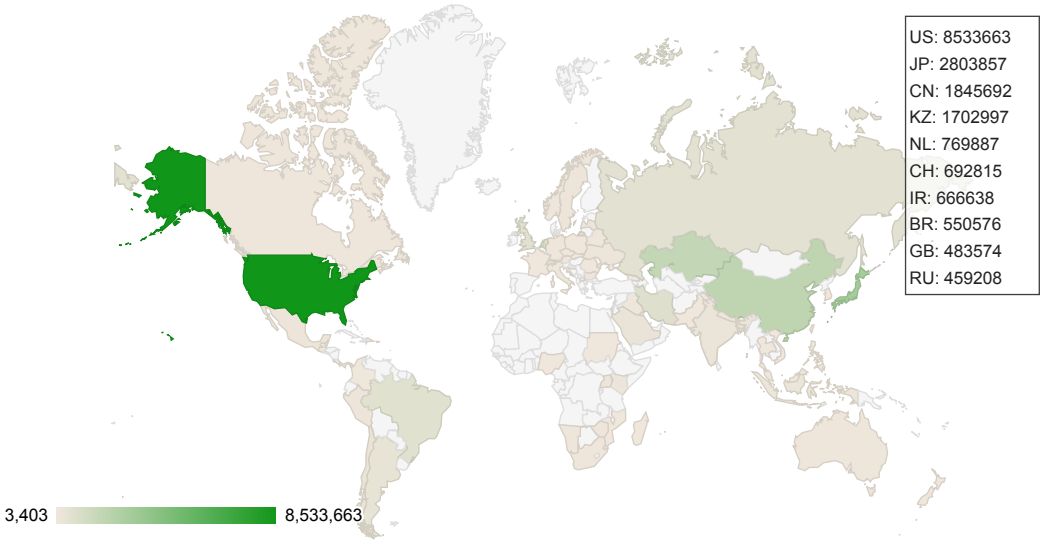


Figure 5.15: Leaks per country

In Figure 5.15 the amount of leaks have been charted per country, where a leak again is only counted when it lasted longer than 1 minute and was of a real valley type. The countries most prevalent appearing at the top leaks per country are mostly the same countries to which the ASes found as being the leak AS in Section 5.5 belong. The United States appear at the top, mostly because of various valleys at Hurricane (Section 5.5.4). Japan appears second, which is mostly influenced by the WIDE route collector data which introduced valleys such as the ones in Section 5.5.1. China appears third which is mostly influenced by the missing siblings relation for aBitCool as described in Section 5.5.2.

Chapter 6

Discussion

From all updates studied and analysed in Chapter 5, about 3.5% was found to contain a "real" valley free violation according to the relation data we used. The type of violation most frequently seen was a peer-to-peer relation followed by another peer-to-peer relation. This most frequently found violation type seemed to be heavily influenced by the top leak triplets. Depending on the route collector used, the valleys detected differed significantly, resulting in different distributions of the violation types between the various collectors.

The majority of the route leaks was found to have very short lifetimes, mostly lower than a minute. This is most likely caused by route convergence. The impact of the routes announced with such a low lifetime is negligible as due to their short duration not much traffic will traverse over this route. About 25% of the valleys however have a longer duration than a minute and about 2.6% lasts even over 6 days. If those are indeed violations of the policies, those leaks will be certain cause for concern.

Over a longer period of time, the amount of detected valleys does not seem to grow. What is found however, is that towards the end of a month the amount of detected valleys increases, while at the beginning of the next month there is a significant amount of valleys that are closed again. This is due to the relations file we use are only refreshed once a month. If a relation thus changes during a certain month it may be incorrectly detected as a valley until the next month were the change in relationship has been incorporated in the new relational data. This indicates that the project will benefit from more frequently renewed relation data.

About 29% of all valleys found in the given time period with a duration longer than a minute, were found to be caused by a group of the 10 most frequently recurring leak triplets. This indicates that certain ASes are more frequently involved in detected valleys than others. However, most of the cases were later on found to be correctly according to policy and thus false

positives, making it uncertain if this also applies for the "real route leaks".

To determine the causes of the route leaks found, several of the frequently recurring leak triplets were further investigated using information we could find about the ASes and the RPSL data if available. For the ASes for which enough information could be found to determine why the valley free violation was detected, it could be determined that the detection was very likely caused by either an incorrectly inferred relationship or a more complex relation than the ones defined in the relationship data we used. For other ASes not enough information could be gathered to determine whether the route leak was correctly detected or not, nor what the cause of the route leak might have been.

Chapter 7

Future Work

7.1 Utilizing knowledge on complex relationships

The inferred AS relation dataset used consists only of provider-to-customer and peer-to-peer relations. The siblings inferences have been added to add the sibling-to-sibling relation over the relation that was originally inferred. However, apart from these three relations there exists also several "complex" relationships. Giotsas et al. [11] have recently published an article about the inference of two of those complex relationships. One of them is the partial transit relationship, where a customer pays the provider for transit to the provider's client and peering routes, but not for its provider routes. The other complex relation that will be inferred is the hybrid relation, where the relationship used between two ASes depends on the interconnection point. This is a type of relation similar to what was previously seen at Section 5.5.4 where the AS had a different relation based on the IP version used. Both of these new types of relation inferences can help reduce the amount of false positives detected. Hybrid relations will allow the program to remove the false positives that were detected because the relation was only inferred for the IPv4 data. Partial transit relationships will likely have been formerly inferred to be a peering relation as it shares most of its characteristics with a peering relation, e.g. the routes from a partial transit client will not be propagated up to a clique AS like in customer-to-provider relations, so the knowledge about these relationships will likely reduce the amount of violations where a peer route is leaked to another peer.

7.2 Adjust relations manually to decrease number of false positives

As seen in Chapter 5, there are quite an amount of valleys found that are most certainly not real route leaks but rather complex relationships to provide connection between research/educational ASes. Therefore to reduce the amount of false positives, the application will benefit from adjusting the relations for these type of ASes manually when such artefacts are found. It was chosen not to adjust the relations manually for the current research as this requires a lot of manual adjustments based on the assumption that these ASes are in fact creating valleys because of those reasons, while other ASes that create valleys because of the same reasons may be missed, because we could not identify them having such relationships. This would create inconsistency in the results, so instead of manually removing them it was decided to document the appearance of them and leave the removal as future work.

7.3 Validation with Autonomous Systems

The majority of the valley free violations currently detected are expected to be either intentional or a result of the complex relations mentioned before. To get a better view on which valley free violations are actually causing trouble and which ones are intentional, validation with the Autonomous Systems responsible for the "leak" is necessary. Based on this feedback the relations inference, siblings inference or working of the system should be modified to better cope with specific situations where false positives are being detected.

Chapter 8

Conclusion

In this project we have used the valley free rule to detect route leaks in publicly available BGP data. We have found and compared two different sets of relation data to find the data set most reliable for this purpose. We have further enhanced this relation data with the siblings inferences from the same source to reduce false positives caused by incorrect inferences that where actually sibling relations. Also we have looked at other related projects to see how we could improve upon them.

We have designed and implemented an application to automatically download publicly available BGP updates, find valleys within them, store them in the database and generate statistics from the data gathered. This application has been used to parse two months of data from 3 different route collectors.

From the data we investigated upon, about 3.5% was found to contain a valley free violation according to the relationships data from CAIDA. A majority of about 66.3% of those valleys was withdrawn from service within the first 64 seconds. The total amount of valleys that have not been withdrawn yet at any given time was found to be dependent on the relation data used, but showed no sign of constant increment nor decrement over longer periods. It was found that the 10 most frequently recurring AS leak triplets were responsible for 29% of all route leaks.

Many of the leak triplets found in the top 10, however were found to be false positives, caused by either incorrect relation inferences or relationships more complex than captured in the relations files used. This indicates that although the tool can filter out possible route leaks from large data sources, it still needs a lot of manual verification to filter out the false positives from the true positives to uncover the "real violations". Utilizing the knowledge on complex relationships and allowing adjustments to the relations may yield better results in the future. Also validating the results with Autonomous Systems will make it possible to better find the cause of violations.

Bibliography

[1] APAN. *the Asia Pacific Advanced Network*. 2009. URL: http://www.apan.net/ (visited on 10/16/2014).

[2] Dan Ardelean and RIPE NCC. *BGPDump*. Jan. 2011. URL: http://www.ris.ripe.net/source/ (visited on 03/26/2014).

[3] Christoph Biedl. *BGPDump Issue #19*. May 2014. URL: http://bitbucket.org/ripencc/bgpdump/issue/19/looking-for-a-security-contact (visited on 07/28/2014).

[4] L. Blunk, M. Karir, and C. Labovitz. *Multi-Threaded Routing Toolkit (MRT) Routing Information Export Format*. 2011. URL: http://tools.ietf.org/html/rfc6396 (visited on 07/28/2014).

[5] Martin Brown. *Pakistan hijacks YouTube*. Feb. 2008. URL: http://www.renesys.com/2008/02/pakistan-hijacks-youtube-1/ (visited on 09/17/2014).

[6] CAIDA. *Mapping Autonomous Systems to Organizations: CAIDA's Inference Methodology*. Aug. 2014. URL: http://www.caida.org/research/topology/as2org/ (visited on 09/17/2014).

[7] Jim Cowie. *Chinas 18-Minute Mystery*. Nov. 2010. URL: http://www.renesys.com/2010/11/chinas-18-minute-mystery/ (visited on 07/30/2014).

[8] Jim Cowie. *The New Threat: Targeted Internet Traffic Misdirection*. Nov. 2013. URL: http://www.renesys.com/2013/11/mitm-internet-hijacking/ (visited on 07/30/2014).

[9] Lixin Gao. "On inferring autonomous system relationships in the Internet". In: *IEEE/ACM Transactions on Networking (ToN)* 9.6 (2001), pp. 733–745.

[10] Phillipa Gill, Michael Schapira, and Sharon Goldberg. "A survey of interdomain routing policies". In: *ACM SIGCOMM Computer Communication Review* 44.1 (2013), pp. 28–34. URL: http://www.cs.bu.edu/fac/goldbe/papers/survey.pdf.

[11] V. Giotsas et al. “Inferring Complex AS Relationships”. In: *Internet Measurement Conference (IMC)*. Nov. 2014.

[12] Vasileios Giotsas and Shi Zhou. “Detecting and assessing the hybrid IPv4/IPv6 as relationships”. In: *ACM SIGCOMM Computer Communication Review*. Vol. 41. 4. ACM. 2011, pp. 424–425. URL: http://conferences.sigcomm.org/sigcomm/2011/papers/sigcomm/p424.pdf (visited on 03/10/2014).

[13] Vasileios Giotsas and Shi Zhou. “Valley-free violation in Internet routing - Analysis based on BGP Community data”. In: *Communications (ICC), 2012 IEEE International Conference on*. IEEE. 2012, pp. 1193–1197. URL: http://www0.cs.ucl.ac.uk/staff/V.Giotsas/files/giotsas.icc.2012.pdf (visited on 03/10/2014).

[14] Google. *Google Charts - Google Developers*. Nov. 2008. URL: https://developers.google.com/chart/ (visited on 07/31/2014).

[15] J. Hawkinson and T. Bates. *Guidelines for creation, selection, and registration of an Autonomous System (AS)*. Mar. 1996. URL: http://tools.ietf.org/html/rfc1930 (visited on 08/25/2014).

[16] Geoff Huston. *Leaking Routes*. Mar. 2012. URL: http://labs.apnic.net/blabs/?p=139 (visited on 07/30/2014).

[17] M. Lepinski. *An Infrastructure to Support Secure Internet Routing*. Feb. 2012. URL: http://tools.ietf.org/html/rfc6480 (visited on 07/30/2014).

[18] M. Lepinski. *BGPSEC Protocol Specification (draft)*. July 2014. URL: http://tools.ietf.org/html/draft-ietf-sidr-bgpsec-protocol-09 (visited on 07/28/2014).

[19] M. Luckie. “Spurious Routes in Public BGP Data”. In: *ACM SIGCOMM Computer Communication Review (CCR)* 44.3 (July 2014), pp. 15–21. URL: http://www.caida.org/publications/papers/2014/spurious_routes_public_bgp/ (visited on 07/28/2014).

[20] M. Luckie et al. “AS Relationships, Customer Cones, and Validation”. In: *Internet Measurement Conference (IMC)*. Oct. 2013, pp. 243–256. URL: http://www.caida.org/publications/papers/2013/asrank/ (visited on 03/04/2014).

[21] Jared Mauch. *BGP Routing Leak Detection System*. Sept. 6, 2007. URL: http://puck.nether.net/bgp/leakinfo.cgi (visited on 02/24/2014).

[22] Jared Mauch. *Detecting Routing Leaks by Counting*. Oct. 2007. URL: http://www.nanog.org/meetings/nanog41/presentations/mauch-lightning.pdf (visited on 07/28/2014).

[23] Riad Mazloum et al. "Violation of interdomain routing assumptions". In: *PAM'2014* (Jan. 2014). URL: http://hal.archives-ouvertes.fr/hal-00926132/ (visited on 03/10/2014).

[24] D. McPherson et al. *Route-Leaks & MITM Attacks Against BGPSEC (draft)*. Apr. 2014. URL: http://tools.ietf.org/html/draft-ietf-grow-simple-leak-attack-bgpsec-no-help-04 (visited on 07/30/2014).

[25] David Meyer. *University of Oregon Route Views Archive Project*. 2014. URL: http://routeviews.org (visited on 02/24/2014).

[26] *MongoDB*. 2009. URL: http://www.mongodb.org (visited on 02/24/2014).

[27] *MongoDB - Geospatial Indexes and Queries*. 2011. URL: http://docs.mongodb.org/manual/applications/geospatial-indexes/ (visited on 07/28/2014).

[28] S. Murphy. *BGP Security Vulnerabilities Analysis*. Jan. 2006. URL: http://tools.ietf.org/html/rfc4272 (visited on 07/28/2014).

[29] *MySQL :: MySQL 5.1 Reference Manual :: 11.5.3.6 Creating Spatial Indexes*. 2005. URL: http://dev.mysql.com/doc/refman/5.1/en/creating-spatial-indexes.html (visited on 07/28/2014).

[30] Jon Oberheide. *jon.oberheide.org - pybgpdump*. 2007. URL: https://jon.oberheide.org/pybgpdump/ (visited on 07/28/2014).

[31] Jon Oberheide. *pybgpdump - Issue 1*. 2013. URL: https://code.google.com/p/pybgpdump/issues/detail?id=1#c6 (visited on 07/28/2014).

[32] Ricardo Oliveira, Walter Willinger, Beichuan Zhang, et al. "Quantifying the completeness of the observed internet AS-level structure". In: (2008). URL: http://irl.cs.ucla.edu/~rveloso/papers/completeness_tr.pdf (visited on 03/04/2014).

[33] Tom Paseka. *Why Google Went Offline Today and a Bit about How the Internet Works*. Nov. 2012. URL: http://blog.cloudflare.com/why-google-went-offline-today-and-a-bit-about (visited on 07/30/2014).

[34] Alex Pisolov and Tony Kapela. *Stealing The Internet - An Internet-Scale Man In The Middle Attack*. Aug. 2008. URL: https://www.defcon.org/images/defcon-16/dc16-presentations/defcon-16-pilosov-kapela.pdf (visited on 07/30/2014).

[35] WIDE Project. *WIDE:About WIDE:Background*. 1988. URL: http://www.wide.ad.jp/about/background.html (visited on 10/16/2014).

[36] Y. Rekhter, T. Li, and S. Hares. *A Border Gateway Protocol 4 (BGP-4)*. Jan. 2006. URL: http://tools.ietf.org/html/rfc4271 (visited on 07/28/2014).

[37] rfc1036. *Parser for zebra/MRT BGP routes dumps*. 2007. URL: https://github.com/rfc1036/zebra-dump-parser (visited on 07/28/2014).

[38] *RIS Raw Data*. 2014. URL: http://www.ripe.net/data-tools/stats/ris/ris-raw-data (visited on 02/24/2014).

[39] *The CAIDA UCSD AS Relationships Dataset*. - <2013-01-01 - 2013-11-01>. 2013. URL: http://www.caida.org/data/active/as-relationships/ (visited on 02/24/2014).

[40] Q. Vohra and E. Chen. *BGP Support for Four-Octet Autonomous System (AS) Number Space*. 2012. URL: http://tools.ietf.org/html/rfc6793 (visited on 07/28/2014).

[41] Stella Vouteva and Benno Overeinder. "BGP Route Leaks Analysis". 2013.

[42] WIDE. *WIDE:About WIDE:Relationship*. 1988. URL: http://www.wide.ad.jp/about/relationship.html (visited on 10/16/2014).

[43] He Yan et al. "BGPmon: A real-time, scalable, extensible monitoring system". In: *Conference For Homeland Security, 2009. CATCH'09. Cybersecurity Applications & Technology*. IEEE. 2009, pp. 212–223.

[44] Yu Zhang. *http://irl.cs.ucla.edu/topology/*. 2013. URL: http://irl.cs.ucla.edu/topology/ (visited on 02/24/2014).

Appendix A

Difficulties experienced

A.1 Unique time values

Although the current MRT specification [4] specifies MRT types for BGP messages containing timestamps with microsecond accuracy (BGP4MP_ET), all of the sources we use dump their data using the BGP4MP MRT type which has only seconds accuracy. When multiple updates for the same prefix are announced by the same neighbour it is important to know which one occurred first, since if the first one contains a valley and the second one does not, the latter update will close the valley created by the previous message. However, it occurs quite frequently that an AS announces the same prefix several times within a second and since the timing is only accurate up till the seconds, it cannot be seen from the individual messages which one occurred first. Therefore to differentiate the updates from the same source, we created an algorithm that adds fictional milliseconds to the timestamps.

- For every update from one neighbour AS all prefixes in that update will be added to a set and the milliseconds index for that neighbour AS will be set to 0

- If another update from that neighbour AS arrives all its prefixes will be compared to the ones already in the set.

- If any of the prefixes was already in the set, the milliseconds index will be upped and the update and every subsequent update will get the increased millisecond set

- If none of the prefixes were in the set, the current value of the milliseconds index will be used.

- When a new second is read, all sets and indexes will be reset.

Just adding a millisecond for every update arrived (by the same neighbour) was also considered, but since ASes sometimes send more than 1000

updates in a single second, this would overflow the milliseconds range. When a single neighbour AS decides to announce the same prefix more than 1000 times in one second, this would crash the program. As of now this has not yet occurred.

A.2 Inconsistent dump filenames

Dump files are usually build at regular intervals. RIPE [38] generates a dump file every 5 minutes, while the dump files at routeviews[25] are generated every 15 minutes. The file name follow an expected format of updates.yearmonthday.hourminute.gz or .bz2. However every once in a while the file is dumped a few minutes later resulting in a filename of for example updates.20140211.0707.gz. Since it sometimes takes quite some time to download the list of all files available at a server in a directory it would slow down the program and increase the load on the server when the filenames would be retrieved from this list all the time. However to make the application compatible with such renames occurring there was a need for some complex scripts that would handle such cases correctly, while minimizing the slowdown for all correct file names.

A.3 Irregular file availability times

Apart from the inconsistency in the filenames, the files are also available at irregular time intervals. This occurs far more often than the inconsistent naming of the files. Normally, the intervals when a new file is available is the same interval as when a file is generated (e.g. 5 or 15 minutes). However, it happens quite often that a file is delayed for a certain time and that afterwards multiple new dump files are made available in a batch. Since the application tries to download and parse the files as soon as they become available, not being able to predict when to download the file can be problematic.

A.4 Python version compatibility

When Python 3.0 was released in 2008, it was decided to clean up the language with no regards to backwards compatibility. Therefore most code written for Python 2.x is not compatible with Python 3.x and vice versa. Since Python 2.x is still the default version in most Linux distributions, it was desirable that the project would be Python 2.x compatible. However,

since this may change in future distributions, the project should be future proof as well. It is possible to keep projects compatible with both Python versions, but for this a lot of workarounds need to be created. Making the print statement / function compatible is pretty easy, as when used as a function in Python 3.x it will just see the parenthesis as redundant in Python 2.x and ignore them. In this section I will describe some of the harder difficulties experienced while making the program Python 2.x and Python 3.x compatible.

A.4.1 Bytes

In Python 2.7 bytes and strings refer to the exact same type. In Python 3.x however, bytes are very different from strings. When retrieving an index from a Python 3 byte, an int will be returned, where in Python 2.7 a str will be returned. This also means that if you want to concatenate two indexes of a byte to another byte object, you cannot just use b[0] + b[1] for example, as this will add the two integers to each other instead. To overcome the differences in bytes in Python 2.7 and 3.x, we have taken two different approaches within the project. When the bytes come from an input stream, we just keep them as string and check for byte operations whether or not the value should be interpreted as string or as integer or as bytes. When we need to handle an argument in a function differently when it is a byte or a string however, e.g. for formatting before inserting into the database, a custom bytes class is used that mimics the behaviour of the Python 3.x bytes type in Python 2.7, while retaining the use of the original bytes in Python 3.x. This way bytes can be differentiated from strings and there is no need for extra workarounds for bytes operations on them.

A.4.2 URLLib

The functions to work with web data in Python 2.7 are scattered around multiple modules. To read the contents of an URL, urllib2.urlopen should be used, as urllib.urlopen is deprecated. To download the contents of an URL urllib.urlretrieve can be used, as urllib2 does not have that function. To urlencode a dictionary urllib.urlencode is used, even though other URL parsing functions can be found in the urlparse module. In Python 3.x there is only one urllib module, which contains the sub modules request, parse, errors and robotparser. To be compatible with both Python 2.7 and Python 3.x, a custom urllib module is created which contains function redirects based on whether or not urllib2 exists. So for Python 3.x the layout will remain the

same, while for Python 2.7 the functions will be in the same sub modules as they are in Python 3.x

It was also decided not to use the original urlretrieve function, because it did not handle errors the way it was expected. When urlretrieve is used to download an URL leading to a non-existing document, the 404 page will be downloaded without warning. It was decided to create a new download function instead which raises an error when an unexpected status code is returned instead, which the program can catch to handle.

A.5 Database

To store all the valleys the program has found a database will be used. There are many different types of databases to choose from, and all of them have different things they can and cannot do. Unfortunately it is not always clear before the beginning of the project what features the database should support and which database should therefore be chosen.

A.5.1 MongoDB

Initially the project used MongoDB. MongoDB is currently the "leading NoSQL database"[26] and is relatively easy to use. Instead of defining a schema beforehand and inserting values based on that schema, you just insert dicts with similar keys to the same collection. For querying also dicts are used with the keys that should be equal or special keys such as $gt for comparisons. It is therefore not necessary to use a specialized language such as SQL to use MongoDB.

However, the amount of valleys added to the database grew very fast and after a while we ran into a series of problems. First of all was the disk space, the server initially had about 15 GB of disk space allocated to the database, but with the indexes used, this was not even enough for 40 days of data.

After expanding the disk space a new issue arrived which involved a scalability issue. The queries used to produce the statistics took over a day, and that was even before all the data originally planned to be used was in the database. After investigating the problem, it was found that some queries could be improved by creating compound indexes. All needed fields were already indexed as single-key indexes, but as the MongoDB version used (2.4.9) did not support index intersection yet, only one index would eventually be used and the rest of the query would be performed without indexes. Adding these compound indexes increased the size of the database

even further, since each type of query now needed its own compound index defined.

On top of that we wanted all queries between a time range of two months, but also wanted certain queries to count the amount of valleys with a duration within a certain range for example. Within a compound index however, only one key in the index can be used for range queries, making it impossible to successfully optimize a query with a time-range and a duration range.

Also to count the amount of valleys active at a certain time point a query is needed in the form time $<$ timepoint $<$ end_time. Now time $<$ timepoint can be indexed or timepoint $<$ end_time, but not both. After investigating into possible solutions, it was found that these type of queries could be made scalable using spatial indexes. However, MongoDB only supports geospatial indexes [27] which did not seem to solve this problem. The spatial indexes that would solve this problem can be created in MySQL [29] and only for MyISAM tables.

A.5.2 MySQL

In order to support queries where valleys are selected that have a time less than a certain value and end time higher than that value, it was decided to rewrite the database part of the application to support MySQL instead of MongoDB. While this indeed allowed the queries to be executed much faster, it significantly incremented the insertion time, as apparently the creation of the values to be indexed is not a quick task. Since the amount of inserts is quite high and the amount of queries to be performed is not that big this solution was thus not sufficient.

Since the queries to be executed to get the data that was necessary for the charts is known beforehand it was decided to, instead of querying the whole database with some keys that are not indexable together, create separate statistics tables which will be filled daily with the results needed for only that day. Since a smaller part of the data is used every time, the lack of some indexes is less noticeable, while when the statistics over a certain period need to be requested only the statistics tables have to be queried and summed together.