Analyzing BGP Policies: Methodology and Tool

Georgos Siganos
Dept. of Computer Science & Engineering
University of California, Riverside
Riverside, USA
siganos@cs.ucr.edu

Michalis Faloutsos
Dept. of Computer Science & Engineering
University of California, Riverside
Riverside, USA
michalis@cs.ucr.edu

***Abstract*— The robustness of the Internet relies heavily on the robustness of BGP routing. BGP is the glue that holds the Internet together: it is the common language of the routers that interconnect networks or Autonomous Systems(AS). The robustness of BGP and our ability to manage it effectively is hampered by the limited global knowledge and lack of coordination between Autonomous Systems. One of the few efforts to develop a globally analyzable and secure Internet is the creation of the Internet Routing Registries (IRRs). IRRs provide a voluntary detailed repository of BGP policy information. The IRR effort has not reached its full potential because of two reasons: a) extracting useful information is far from trivial, and b) its accuracy of the data is uncertain.**

In this paper, we develop a methodology and a tool (Nemecis) to extract and infer information from IRR and validate it against BGP routing tables. In addition, using our tool, we quantify the accuracy of the information of IRR. We find that IRR has a lot of inaccuracies, but also contains significant and unique information. Finally, we show that our tool can identify and extract the correct information from IRR discarding erroneous data. In conclusion, our methodology and tool close the gap in the IRR vision for an analyzable Internet repository at the BGP level.

I. INTRODUCTION

The overarching goal of this work is to model and improve the robustness of the Internet at the BGP level. The Border Gateway Protocol [1] is the protocol that dictates routing between Autonomous Systems (AS), and implements their business policies. The importance of BGP has become clear in the network community over the last five years, and several efforts have improved our understanding of BGP [2] [3] [4]. However, we still have a long way to go: studies show that BGP operates in a far from robust state and many of its behaviors are not well understood. The need for a robust Internet has created efforts like the Internet Routing Registries (IRR), a distributed database, where ASes store their policies. However, IRR has not reached its potential nor fulfilled the initial vision [5]. Our work attempts to take the IRR to the next level. We provide a systematic approach and a tool, Nemecis, to extract and infer useful information from IRR, with the ultimate goal to use this information to model, manage and protect Internet routing.

There exist a number of tools to measure actual BGP routing, like ping, traceroute, looking glass, BGP table dumps. But there does not exist a tool to bridge the gap between intended policy (configuration) and actual routing. Internet Routing Registries (IRR) [6], contain the policy of a large number of networks, expressed in a high level language, RPSL [7] [8]. These registries are considered by a lot of people to be useless and outdated, based primarily on empirical evidence. To the best of our knowledge, there does not exist a tool that can analyze these policies, and check their validity or freshness. The registries are maintained manually and in a voluntary basis to a large extent, and the policies remain as simple text. Thus, analyzing IRR is not a trivial task. The difficulties lie in: a) RPSL is very flexible, so policies can be very complex, b) there can be many different ways to express the same policy, c) the registries can contain inaccurate, and incomplete data. At the same time, the information of IRR is important in order to understand and interpret Internet Routing, since Routing tables are not sufficient to understand the intended policies.

Despite recent efforts, we do not have a complete model for BGP, and the robustness problem has not been solved. First, several measurement studies show that the Internet behavior is not fine-tuned [9] and it can be destabilized through cascading effects [10]. Second, Internet routing relies on a large part on trust. Networks usually assume that the information they receive from their neighbors is correct. This is not always the case, since misconfigurations, bugs in the software of the routers [11] and malice [12] are common. Third, there have been very few (public) efforts to automate monitoring and management of BGP [13] [14]. Finally, the issue of secure and robust Internet routing is an open problem despite significant research efforts and studies [15] [16] [17] [18].

How can we automate the management and the safety of the Internet? This is the general problem we attempt to address. Within this framework we focus on three major problems in BGP research and its management: a) the lack of detailed knowledge and of accurate models for the Internet at the BGP level, b) the lack of tools to analyze the configuration of an Autonomous System, and to check whether the registered policy matches the intended policy, c) the need for an automatic way to detect abnormal routing behavior. Checking for errors is a tedious manual process, usually done in a reactive way.

In this paper, we develop a methodology and a tool for addressing the issues we just described. We call our tool Nemecis, which stands for NEtwork ManagEment and ConfIguration System. Our goal is to provide a framework for

This material is based upon work supported by the National Science Foundation under CAREER Grant No. 9985195, DARPA award NMS N660001-00-1-8936, a grant by TCS Inc., and DIMI matching fund DIM00-10071.

the analysis of RPSL policies, which can be used during the configuration phase, or the operation phase. During the configuration phase we can check the registered policy for correctness. During the operation phase, we can check whether the intended policy matches the actual routing. This way, we can reduce the time it takes to discover and fix routing problems. Most importantly, we can start to monitor how Internet routing works. In fact, our tool is among the first public tools to analyze the IRR policies. RIPE, has as a long-term goal to validate the policies that Autonomous Systems register, and thus increase the robustness of BGP. Our work here is the first step in reaching this ambitious goal. Our contributions can be summarized in the following points:

- We provide Nemecis, an efficient tool to analyze the IRR/RPSL information. Our tool can be used to parse, clean and infer the business relations found in the Internet Routing Registries, and create an easy to query relational database, where the policies are stored in tables and not as simple text.
- We validate the accuracy of our model and tool by comparing the results from IRR with real routing tables. Our accuracy of inferring correctly the policy is higher than 83%, which we consider to be very good, if we take into account the quality of the registered policies.
- We quantify the usefulness of the IRR information: we find that 28% of the ASes have both a consistent policy and are consistent with BGP routing tables. Note though that almost all are from one only registry, RIPE.
- We identify commons mistakes and problems in IRR registries. We discuss ways to overcome them so that IRR can be used to automate the management and safety of the Internet routing.

Our work in perspective. Does it make sense to analyze IRR when the information is to a large extent inaccurate? Our answer to this is twofold. First, we reverse the argument: if we have tools to use IRR effectively, then network administrators will be motivated to keep IRR accurate and up to date. In addition, the tool can be used even with locally correct information: between neighboring ASes. Thus, it does not require global conformance. Second, we claim that the robustness and ultimately the security of the Internet will need global coordination and conformance to timely updates. The security of an interconnected system is equal to the security of its weakest component. To achieve such a goal, we need a) compliance and information, and b) methodologies and tools. Our work covers the second part, continuing the vision of IRR [5] for an analyzable and robust Internet.

The rest of this paper is structured as follows. In section II we present some definitions and background work. In section III we describe our framework. In section IV, we analyze the IRR registries, with a twofold goal, to validate our tool, and to check the status of the registries. In section V we discuss how we can use our methodology to automatically detect misconfigurations. In section VI we present our conclusions.

TABLE I
WHAT DOES AN AS EXPORT TO ITS NEIGHBORS?

	Provider	Peer	Sibling	Customer
Provider			√	√
Peer			√	√
Sibling	√	√	√	√
Customer	√	√	√	√

II. BACKGROUND AND PREVIOUS WORK

In this section, we briefly describe an overview of Internet routing, Internet routing registries and the language used to describe the routing policy. Additionally, we discuss some of the previous work that either relate or provide a motivation for our paper.

A. Internet and BGP-4

Internet is structured into a number of routing domains that have independent administrations, called **Autonomous Systems (AS)**. Each autonomous system is identified by a number, **asn**, which is assigned to it by an Internet registry. An Autonomous System uses an intra-domain routing protocol, like OSPF or IS-IS, inside its domain, and an inter-domain protocol to exchange routing information with other Autonomous Systems. The defacto standard for inter-domain routing is **BGP-4** [1]. The primary difference between the intra-domain and the inter-domain protocol is that the first one is optimized for performance, solely based on operational requirements, while the second is used to enforce the **policy** of the Autonomous System, which corresponds to the **business relations** with its neighboring ASes.

An Autonomous System given its policy, will advertise to its neighbors a list of **IP Prefixes**, or **routes** that are reachable through it. Each route is tagged with a number of **attributes**. The most important attribute is the **AS_PATH**. The AS_PATH is the list of ASes that packets towards that route will traverse. The **Community** attribute is a 32 bit number that is usually used to influence the routing of a provider.

An AS uses **filters** to describe what it will import from and export to a neighboring AS. The filter can include a list of routes, a list of regular expressions on the AS_PATH, a list of communities, or any possible combination of these three. Filters can have both positive and negative members. For example we can explicitly reject routes that are either private [19], or reserved [20].

B. Business Relations

In the literature [21] [22] [23], there exist four basic types of business relationships among ASes. The **Provider to Customer and Customer to Provider** type. The customer AS buys transit access to the Internet from a provider AS. The provider advertises either its full routing table, or a default route to the customer, and accepts from the customer its local routes and the routes of its customers. Another type is the **peer to peer** links in which the ASes exchange their local routes. This way they don't need to go through their providers in

order to reach each other or their customers, for economic and performance reasons. Another type is the **sibling to sibling**. An ISP can own more than one AS number. Each can serve a specific purpose, like represent the backbone network, or a region, for example Europe. In this case these ASes advertise to each other all the routes they learn from their neighbors. In Table I, we have a brief overview of what an AS exports to its neighbors based on their business relation. The basic observation from the table is that we can group the relations into two basic categories. In the first category, we have the policy towards Providers and Peers where an AS restricts what it exports. In the second category, we have the Customers and the Siblings where an AS gives unrestricted access. The relations we just described, are the typical relations used in the literature. More complex policies can exist, variations of the simple policies we described above.

C. Internet Routing Registries

The need for cooperation between Autonomous Systems is fulfilled today by the **Internet Routing Registries (IRR)** [6]. ASes use the **Routing Policy Specification Language (RPSL)** [7] [8] to describe their routing policy, and router configuration files can be produced from it. At present, there exist 55 registries, which form a global database to obtain a view of the global routing policy. Some of these registries are regional, like RIPE or APNIC, other registries describe the policies of an Autonomous System and its customers, for example, cable and wireless CW or LEVEL3. The main uses of the IRR registries are to provide an easy way for consistent configuration of filters, and a mean to facilitate the debugging of Internet routing problems. Unfortunately, the tools that exist today for the purpose of verifying the policy [24], are based on visual inspection of the policies. As we mention earlier, there does not exist a systematic way to check for consistency of the registry. Additionally, most of the policy is stored as text in a database, and there exist a small number of predefined queries that can be executed, compared to the wealth of information that is contained in the database.

D. Routing Protocol Specification Language(RPSL)

RPSL is the language used in IRR to specify the routing policy of an AS. The design goal is twofold. First, RPSL provides a standard, vendor independent language, so that the policy of an AS can be published in an easy to understand format. Second, RPSL provides high level structures for a more convenient and compact policy specification. RPSL provides an abstract representation of policy, but still the policy described is based on filters on routes, on regular expressions on the AS_PATH, and on communities.

There exist 12 different classes of records, that either describe portion of a policy, or describe who is administering this policy. In figure 1, we have a simplified definition of the four more important classes used to describe the policy. The route class is used to register the IP prefixes or routes an AS owns and originates. The as-set and route-set classes are high level structures that can be used to group routes. For example

```
The route class
route:        <IP prefix>
origin:       <AS number>
member-of:    list of <route-set-names>
mnt-by:       list of <mntner-names>

The as-set class
as-set:       <object name>
members:      list of <AS numbers> or <as-sets names>
mbrs-by-ref:  list of <mntner-names>
mnt-by:       list of <mntner-names>

The route-set class
route-set:    <object name>
members:      list of <IP prefixes> or <route-set-names>
mbrs-by-ref:  list of <mntner-names>
mnt-by:       list of <mntner-names>

The aut-num class
aut-num:      <as-number>
as-name:      <object name>
import:       from <peering> [action <action>]
              accept <filter>
export:       to <peering> [action <action>]
              announce <filter>
default:      to <peering> [action <action>]
              networks <filter>
mnt-by:       list of <mntner-names>
```

Fig. 1. A simplified version of the main classes of records in RPSL

an AS can create a route-set that will contain the routes of its customers. Finally, the aut-num class contains the import and the export policies for every neighbor of the AS. The policies are expressed in the form of a filter. Note that every class has a mnt-by attribute that specifies the maintainer of the record. This is done for security reasons so that only the maintainer can update that record. There exist additional attributes, not shown in the figure, like the source attribute that specifies in which registry the record exists, and the changed attribute that provides the date that the record was either last updated or created[1]. Next, we describe in more details the various ways that filters can be specified in RPSL.

Filtering based on Routes. The most common way to describe the policy of an AS is to determine the specific routes that the AS will import from an AS and export to that AS. There are many ways to express this in RPSL. First one can directly use routes, for example *"from AS3 import {199.237.0.0/16}"*. For convenience, RPSL allows routes to be grouped. One way is to use *AS numbers*, like *"to AS2 announce AS3"*, that means export to AS2 all routes that AS3 registers. RPSL allows also for explicit definitions of groups called **sets**. There exist two different types of sets, the **as_sets**, and the **route_sets**. The first contains other as_sets and AS numbers. The second contains other route_sets and routes.

Filtering based on regular expression on the AS_PATH. Another way to express policy is to use the AS_PATH attribute of the advertisements. An example is *"from AS3 import* $<$ *^AS3+ AS5*$*$\$ $>$". This regular expression can match paths like "AS3" and "AS3 AS5". The regular expressions are POSIX compliant, and can include AS numbers, as_sets, AS number

[1]Note that the changed attribute is not updated in an automatic way by the registry, but is provided by the maintainer of the record. This can lead to cases where the record is changed but the time is not updated.

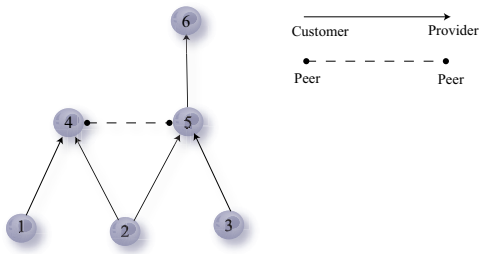

Fig. 2. A simple multihoming example, where AS2 has two providers AS4 and AS5.

sets, unary operators like *, and binary operators. This way of expressing policy is convenient in the sense that ASes don't need to update their filters, if a neighbor AS gets a new route. But, it allows for misconfigurations errors to travel further to the core of the network, as for example in the case where an AS by mistake advertises a route it does not own.

Filtering based on communities. The third way to express policy is to use communities. Communities can be thought of as a mechanism to group routes, and treat the group as one entity. This grouping is often referred to as "coloring". For example, all routes from customers can be colored with community *'112:10'*, and then we export the routes based on the communities. Interestingly, we find that this is very rare in IRR, only 21 ASes or 0.2% of all ASes use communities in their filters, and we will not consider it in our analysis.

Keywords used in the filters. There exist a number of keywords in RPSL that can be used to describe a policy. There can be keywords like *'ANY'*, that describe any route that the AS receives. The keyword *'PeerAS'* is often used with sets, and it facilitates looping over all members of a set. We also have keywords like *'refine'* and *'except'*, which can be used to describe the policy in a more compact way. We consider these details to be out of scope of this document. The interested reader can find more details in the RFC2622 [7].

E. Previous Work

We did not find any effort to analyze IRR in the detailed fashion we do here. However, several related studies exist.

Chen et al. analyze the IRR in [25]. Their objective is to discover the AS level topology of the Internet, and so they are interested in the existence of links between Autonomous Systems. They use a number of tests to check whether the records are up to date. They examined the RADB and RIPE registries and find that the RIPE registry contains more accurate topological information. Note though, that they don't analyze the registered policies, which is our focus.

Feldmann and Rexford [26] examine our ability to validate the correctness of the configuration of routers within an ISP. They describe a case study of the AT&T IP Backbone and the problems they face analyzing the configuration files.

Using a high level policy language to produce configurations can significantly reduce the number of errors. Ratul et al. [11] find that a significant portion of the instability observed in the Internet today is due to misconfigurations. They find that 3 out of 4 new prefix advertisements are the result of misconfiguration. Additionally, between 0.2 and 1.0% of the BGP table size suffers from misconfiguration daily. Misconfigurations can occur either by mistake or malice, for example a spammer can hijack a route temporarily [27].

```
as-set:     AS-5
members:    AS5, AS5:AS-CUSTOMERS
mnt-by:     AS5-MNT

as-set:     AS5:AS-CUSTOMERS
members:    AS2,AS3
mnt-by:     AS5-MNT

as-set:     AS4:AS-CUSTOMERS
members:    AS1,AS2
mnt-by:     AS4-MNT

route:      199.237.0.0/16
origin:     AS5
mnt-by:     AS5-MNT

aut-num:    AS5
import:     from AS6 action pref = 100; accept ANY
import:     from AS4 action pref = 90;
                 accept <^AS4+ AS4:AS-CUSTOMERS*$>
import:     from AS2 action pref = 80;  accept AS2
import:     from AS3 action pref = 80;  accept AS3
export:     to AS6 announce AS-5
export:     to AS4 announce AS-5
export:     to AS2 announce ANY
export:     to AS3 announce ANY
mnt-by:     AS5-MNT
```

Fig. 3. Example of RPSL policy for Autonomous System 5

Griffin et al. [28] describe the need for a new policy configuration language. They mention that the complexity of routing policies has grown significantly, and policy interactions can be very difficult to predict. They discuss the requirements and the design space for new configuration languages.

III. Framework for IRR analysis

We develop a framework to analyze IRR registries. Our framework is designed to cope with incomplete and incorrect data, and to be general enough to infer the policy of any Autonomous System without requiring a specific way of policy registration. We first present an overview of our framework and then discuss in details its three main steps.

A. Problem Definition and Overview

The following problem lies in the heart of our effort.

Problem: Given an AS A, its neighbors peers, and the RPSL records that describe the policy of A and the policy of its peers infer the business relations of A [2].

In order to infer the business relations correctly, we have to solve two sub-problems:

- We need to convert the policies using filters to an equivalent **link-level policy**. In the link-level equivalent policy, we replace the export and import filters, with a boolean matrix that describes the relation between the

[2]Note that in the definition we require that we know the policy of its neighbors in order to find the business relations. Even though in most of the cases we can check the policy for correctness in isolation, we can find the business relations more accurately if we know the policy of its peers.

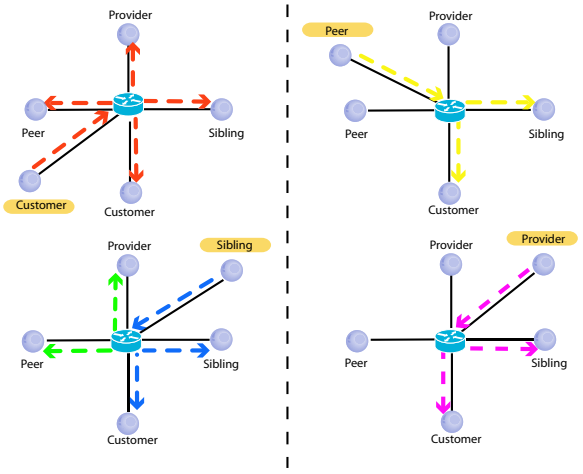

Fig. 4. Import/Export based on the business relationships.

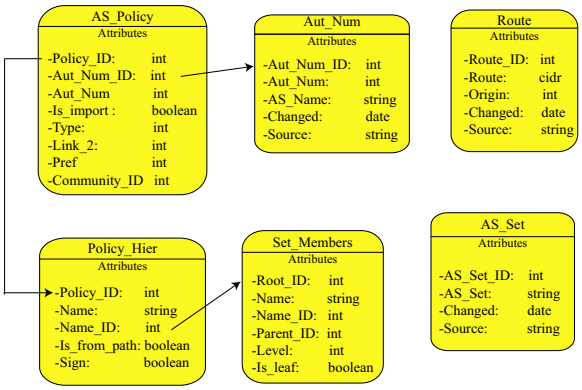

Fig. 5. Part of the schema we use to analyze the policy.

links for an AS. For example, if we import a route from link i, and export that route to link j, then the value of the matrix at (i, j) will be true. In figure 4, we provide a visualization to better understand the link-level approach. For our basic types, we show that the actual policy is to import something from a link and forward it to another link. By converting the problem to the link level, the problem becomes independent of the different kinds of implementations of the policy, or about specific routes or sets we export. This way we concentrate on how to model the actual policy.

- The second sub-problem is to infer the business policies using the link-level model. This sub-problem is independent of the first one. For example, we can enrich the business relations to include more types of relations that we don't consider now, such as backup links, without changing the link-level approach.

Throughout this section, we will use the simple scenario shown in figures 2 and 3 to illustrate the problems we face. The interesting part of this particular setup is that AS2 has two providers, AS4 and AS5.

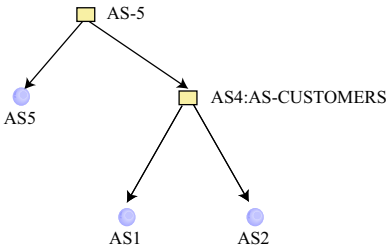

Fig. 6. The directed tree representation for the as_set AS-5.

B. Step One: Building the database

The purpose of the first phase is to parse the RPSL definitions, and populate the tables in the database. In figure 5, we have part of the schema that we use to store the policies. For example, for the table AS_Policy the values that we store for every row are the AS number in the aut_num attribute, the unique id that we give to this AS in Aut_Num_ID[3], the id of the filters in policy_id, which references table Policy_Hier. The import filters can be identified by Is_import=='True' and we also keep the neighbor AS in Link_2. This table and the Policy_Hier table are the main tables used to store part of the policy of an AS. With the exception of the aut_num record, it is straight forward to load the data in the database. For every class of records, RPSL specifies a number of mandatory and optional attributes similar to a SQL table. For these cases, we create the tables with the same attributes and load the data.

We develop a uniform way to analyze filters, which is a critical part of inferring the policy.

Converting as_path filters to route-based filters: Analyzing filters is easy if the filter is a simple string like *"AS1 AS2 AS3"*. However, it can be challenging if the filter is a regular expression like $<\hat{}$AS3+ AS5$*\$>$". In our approach, we analyze the regular expression on the AS_PATH, and we find all the paths that can be accepted. The last asn or as_set in these paths will be used to describe the filter. This way we have an equivalent, but more strict, route-based filter for every path-based filter. In table Policy_Hier, we have a boolean attribute, Is_from_Path, which can be used to identify these cases.

Converting sets to directed trees: The definition of sets in section II allows a set to specify another set as its member. Sets, either as_sets or route_sets, can therefore be modeled as a directed graph $G_S = (V, E_S)$. A directed edge $e = (u, v) \in E_S$ exists, if and only if, v is a member of the set u. For a more simple format, we convert the sets to directed trees. From the graph G_S, we compute for every node $Q \in V$, a directed tree $T_S(Q) = (V_T, E_T)$[4].

In figure 6, we have an example of a directed tree. The directed tree is for the as_set AS-5, its RPSL records are in figure 3. Note that the non-hierarchical objects like $route$ and

[3]We can have more than one aut_num record for an AS, but only one per registry. In our analysis we use the more recent one.

[4]The graph can contain cyclic paths, that is, there can exist sets that are members of each other. We use a strongly connected components algorithm [29] to identify these cases and compute the members of the tree.

asn will also be modeled as directed trees, where there exists only the root of the tree. Using this methodology we populate the Set_Members table.

For the rest of the section we will use the following definitions. We use the term **object** to refer to the directed trees used in the filters. Given an AS A, we use the notation $Import\_From(B) = \bigcup object_i$ to describe the set of objects that the AS is importing[5] from a neighboring AS B. Similarly, $Export\_To(B)$ is the set of objects that AS A exports to AS B.

C. Step Two: Find Link Level Equivalent Policy

In this step, we want to find the relationships between ASes and to do that we analyze the import-export behavior at the level of a link. We will say that an AS A **exports a link** i **to a link** j, if A exports to link j whatever it imports from link i. We want to find this relationship between every pair of links of an AS, for every AS.

More formally, we define the $LinkLevel$ matrix as a NxN boolean matrix, which captures the relations between imports and exports, where N is the number of neighbors of the AS.

Problem: Given an AS A, and its $Import\_From$ and $Export\_To$ sets, compute the boolean matrix $LinkLevel[i, j]$, such that $LinkLevel[i, j]$ is true, if and only if $Export\_To[j] \supseteq Import\_From[i]$.

We describe our method at a high level of abstraction. First, we select each AS and analyze the relationships between its links. We examine each object the AS exports.We find whether the object is imported or constructed by the AS. This way, we find the origin of each exported object. Finally, we find which links are exported to each link.

We present our approach in some more detail to show some interesting challenges. We will start by providing the pseudocode of the procedure that matches the import with the export objects.

```
. Procedure MatchImportExport()
.    for every AS A in the database
.       for every neighbor, B, of A
.          for every object X that A exports to B
.             procedure FindOrigin(X)
```

The procedure FindOrigin() finds whether X is imported and from which links. Algorithmically, the procedure can be described as follows.

```
. Procedure FindOrigin(X)
.    for every neighbor C of A
.       for every object Y that A imports from C
.          procedure compare(X,Y)
```

The procedure compare(X,Y) will identify whether object X is the same as object Y. If it is, we consider that this neighbor could be the origin of object X.

[5]Special care should be given to the case that we have the keyword 'ANY' as an import filter. There exist cases that an AS exports an object without explicitly registering it in any import filter, but the object is implicitly included from an import 'ANY' rule. To solve this problem, we find for every import 'ANY' rule whether the other AS registers the reverse direction of the link. If it does and uses an object other than the keyword 'ANY', we add it to the $Import\_From$ set.

The above procedure hides several subtleties. We highlight the most important ones here.

1) The object X may be a set. In such a case, we want to examine each member of X and identify where each member is imported from. Each member of X can be a set as well, so this procedure continues recursively. However, we only need to examine these recursive definitions only up to a point. Namely, when examining a particular AS, we only refine its sets to components that are "visible" to that AS.
2) An object X may be imported from several neighbors. We would like to disambiguate such cases.

Let us provide some more explanation and our solutions to these issues.

Definition of **Set Construction:** The sets an AS uses can have two origins, they can be: a) defined by the same AS, and b) imported from another AS. When an AS defines a set we say that it **constructs** it. We make the reasonable assumption that every set is constructed by one AS.

For example, consider the scenario in figures 2 and 3. AS5 constructs[6] the as-set *AS5:AS-CUSTOMERS* to describe its customers, which includes its neighbors AS2 and AS3. Similarly, AS4 constructs *AS4:AS-CUSTOMERS*. AS5 imports *AS4:AS-CUSTOMERS* from AS4, and therefore can use it, i.e., export it. Note that *AS4:AS-CUSTOMERS* contains AS2 and AS5 imports AS2 directly. In some sense, AS5 appears to import AS2 twice, one directly and one indirectly through AS4.

Let us elaborate more on the previous example to motivate on our subsequent work. Consider that AS5 exports AS2 to AS6. When our algorithm will examine AS5, it will examine the origin of AS2 that AS5 exports. The algorithm will find that AS2 can be imported from two sources, as we mention above. This can have undesirable effects. To illustrate this, we change the original scenario in figures 2 and 3 and we remove AS2 as a customer of AS5. This means that AS5 does not export AS2 to AS6, since AS2 is not a member of *AS5:AS-CUSTOMERS*. Additionally, there are no import and export rules for AS2. The next step is to consider that AS2 becomes a customer of AS5. Consider that AS5 updates the *AS5:AS-CUSTOMERS* object to include the new customer, but neglects to update its aut_num record. When our algorithm will examine AS5, it will examine the origin of AS2 that AS5 exports. The algorithm will find that AS2 can be imported through AS4. In this case, we will consider the policy of AS5 to be correct, while it is clear that it is not, since the aut_num record misses the import and export rules for AS2. Next, consider the same scenario with medium to large ISPs that have a large number of connections and multiple paths for every AS. How can we check for the correctness of their policy when they import objects that can contain thousands ASes? We provide a systematic methodology to do this below.

[6]To be precise, the maintainer of the AS5 will define the set AS5:AS-CUSTOMERS.

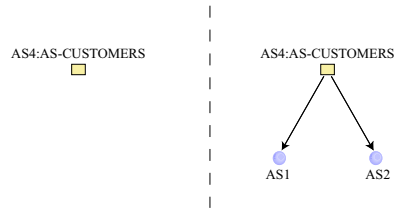

Fig. 7. Set construction example. In the left side we have the directed AS-visible tree as seen in AS5 and in right as the AS4 sees it.

Definition of **directed AS-visible tree:** To accurately compute the link-level model, we need to restrict the depth in which we analyze an object. Intuitively, when examining an AS, we want to analyze the objects involved only for the members that the AS constructs or imports directly. We introduce the notion of directed AS-visible tree. For an object Q, its directed tree $T_S(Q) = (V_T, E_T)$, and an AS A, we define the directed AS-visible tree $T_{RT}(Q, A) = (V_{RT}, E_{RT})$, as a new tree which has the following properties: a) $V_S \supseteq V_{RT}$ and $E_S \supseteq E_{RT}$, and b) an edge $e = (u, v) \in E_{RT}$ exists if and only if, AS A constructs u. We find the new tree by starting from the root of the tree and going one level deeper, if the AS constructs the current node.

The basic idea is that in analyzing the policy of an AS A, the detail we want for a set Q depends on the AS A that uses it in its filters. The following example will illustrate our approach and is based in the scenario in figures 2 and 3. In figure 7, we show how AS5 and AS4 "see" object *AS4:AS-CUSTOMERS*, i.e., T_{RT}(*AS4:AS-CUSTOMERS, AS5*) and T_{RT}(*AS4:AS-CUSTOMERS, AS4*). AS5 imports *AS4:AS-CUSTOMERS*, so its members are not "visible". This way, when it tries to match the import with the export objects, it won't find that it can also reach AS2 through AS4.

Full link constraint: So far, we have introduced a methodology to process the objects in isolation. We also need to consider the cases that an AS exports more than one object to a link. In this case, we use the following rule: We say that an AS A exports link i to link j, if and only if all objects in $Import\_From[i]$ are in $Exported\_To[j]$.

Let's assume that AS5 declares its policy to AS4 in the following way:

```
import:     from AS4 action pref = 90; accept AS4 AS1 AS2
```

The FindOrigin() procedure should not find that AS2 is imported through AS4. If we do, we will have the same problems that we described earlier.

Refinement through multiple executions. If the IRR contained perfect policy definitions, we could easily find for every object the AS that constructs it. Additionaly, we would only need to run the MatchImportExport() procedure once, and we would find the LinkLevel matrix. In practice, identifying the constructor AS is far from trivial, and sometimes we need to consider more than one candidates in order to find which AS constructs the object. To overcome this, we follow a trial and error approach. We make a guess for the constructor. Apply MatchImportExport() once, and examine the results. This can lead us to execute MatchImportExport() once more to identify the constructors. Having the constructors with some degree of confidence we execute MatchImportExport() to find the link-level equivalent policy. Clearly, there are a lot of details that we simply cannot explain in the limited space.

The algorithm that we use to compute the LinkLevel matrix is the following.

. Algorithm for finding Link-Level Equivalent Policy
. A. Identify Candidates ASes for construction
. B. MatchImportExport()
. C. Examine legitimacy of Candidates
. D. Goto step B and repeat one time
. E. MatchImportExport()

We explain each process by referencing to it by letter. Note that MatchImportExport() has already been discussed.

A. Initial step on finding candidates for set construction:

1) For every object X find all the ASes A such that $maintainer\{X\} == maintainer\{A\}$. These ASes are possible candidates for construction for the object X.
2) For every remaining object, we try to guess its constructor. For every AS A, we have a counter that counts how many times other ASes have imported this object from A. We choose the AS that has at least 10 times more counts than the other candidates. For the rest of the objects, we just use all the ASes as possible candidates.

C. Examine legitimacy of Candidates. The result of the MatchImportExport() procedure depends heavily on how we identify the AS that constructs an object. In this step, we evaluate the original selection. We have two cases, in the first the object has one candidate only, in the second it has more than one.

In the first case, we check the number of members of the object that we can match in the policy of the candidate AS. If we can find all the members, then we consider the selection correct. Additionally, if we miss only one member of the object, but we know at least three, we consider the selection correct. For the rest of the objects, we use the approach of step two of procedure A to find one or more additional candidates. We use both the original and the new candidates, and we re-run the MatchImportExport() for these sets.

In the second case, we pick the candidate with the largest number of matched members in its policy. This AS will be the only one that constructs the object.

It is worth mentioning an interesting property of LinkLevel matrix. LinkLevel must be symmetric: LinkLevel must be equal to LinkLevel^T. This is true for the business relations we have defined. For example, if we advertise a customer to a provider, it makes no sense not to advertise the provider to the customer. This property provides a sanity check: if the Customers/Siblings of an AS are correct, then the symmetry must hold.

D. Step Three: Finding Business Relations

In this step, we want to infer from the link-level policies the business relations. Business relations can be grouped by the export filters, as we mentioned in section II. In the first

TABLE II
BUSINESS RELATION BASED ON THE EXPORT CATEGORIES

Business Relation	First AS	Second AS
$Provider \longrightarrow Customer$	A	B
$Customer \longrightarrow Provider$	B	A
$Peer \longrightarrow Peer$	A	A
$Sibling \longrightarrow Sibling$	B	B

category, we have the providers and peers, we will call it *category A*, where we export only the local routes and the routes of the customers. In the second case, *category B*, we have the siblings and customers, where we export everything from all the neighbors. If we assume that all the Autonomous Systems register their policy, we can find the business relation by comparing the category we have with the category of the neighboring AS, as shown in Table II. For example when an AS gives full access to all links, while in the reverse direction the other AS gives restricted access, the link is of type Provider to Customer.

In some cases, we can't find the export policy of the neighbor AS. We need to rely on the policy of one AS only. The idea we use is that if an AS imports ANY from its neighbor, we will have category B, else it will be category A. We can improve the inference if we can identify one of our Providers. We can do this safely if there exist a default 'ANY' rule in the policy of the AS. If there exists an export rule for this link, we know that the links associated with the objects in the filter are either customers or siblings. The remaining links will be either provider or peers.

Limitations of looking only at one AS's policy. The reason that we use the export filters is that we can not always infer the policy of an Autonomous System by just considering its own policy registration. For example, we can have a default free customer to provider relation. In this case, the AS imports from its provider a set. This case has no difference than a typical peer to peer relation, from the point of view of the local AS. Another example is when an AS doesn't filter what it import from a peer. In this case, the AS will use the filter 'ANY', it will accept anything the other AS sends. The result is that we will consider the neighbor AS as a provider instead of a peer.

The only possible way to correctly infer the business relations using only the local AS, is to check the preference that the local AS gives to the filters per link. The order of preference should be customers, peers and then providers. This mean that an AS will always prefer first the path to the customers, then the peers and last the providers. Unfortunately, this is not always true, we can find a lot of cases where either we have the same preference for all the links, or we have the reverse preference, and currently we don't use it.

IV. TOOL AND FRAMEWORK VALIDATION

To evaluate the performance of our framework, we use the IRR registries of *June 22, 2003*. There exist 55 registries,

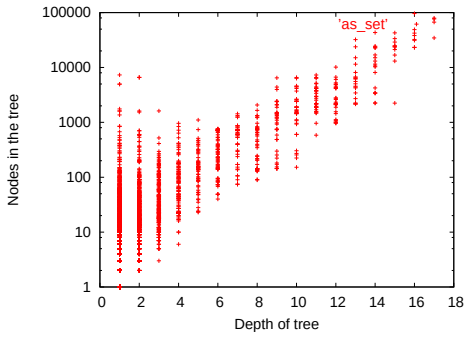

Fig. 8. The number of nodes versus depth for the as_sets.

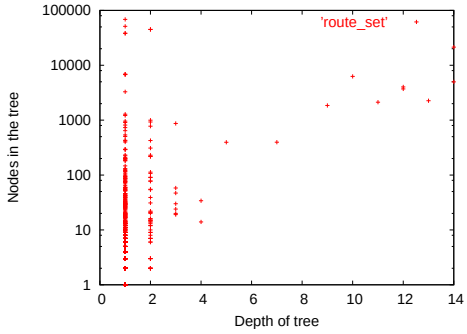

Fig. 9. The number of nodes versus depth for the route_sets.

which are mirrored by RADB [30]. Our tool extracts the information stored in these registries. We validate the information we extracted with Internet routing tables we collected at the same day from Routeviews [31].

Our tool Nemecis, takes as an input a number of Internet Routing Registries, and returns a complete database, where the policies are stored in tables, so that queries can be asked on the policy. Additionally, it returns a list of all errors found during the processing of the registries. Finally, our tool infers the business relations of the Autonomous Systems. We implemented our tool using python and postgresql SQL.

The goal of this section is twofold. First, we want to show that our tool can be used to check the consistency of the registries, detect and discard errors, and that we can extract correct information. Second, we show that the IRR registries, especially the RIPE registry, contain some useful information, but there is a need for processing and cleaning.

A. Building the database

In phase one, we parse and unify all the 55 IRR registries, in a single database. We process every record that can be used in a policy. In total, we have $10,841$ Autonomous Systems, $211,528$ routes, $4,923$ as-sets, and $1,134$ route-sets. The problems in this phase, range in complexity. Some of the problems we faced are the following. First, we have simple problems like routes that are not valid IP Prefixes, dates that either don't follow the correct format or don't exist in the calendar, and more serious, like objects that are referenced in the policy but are never registered. For example, there exist

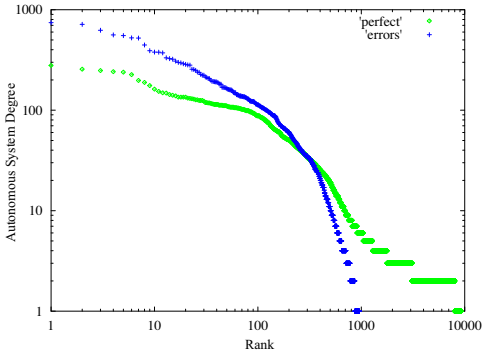

Fig. 10. Degrees for ASes that can be transformed correctly and incorrectly.

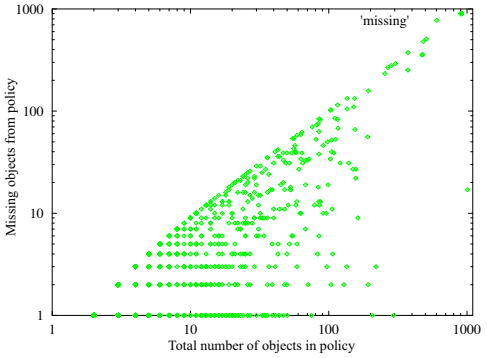

Fig. 11. Number of missing versus the total number of objects in the policy of an AS.

$1,032$ as-sets that are referenced, but don't exist in any of the registries[7]. Additionally, if the objects don't exist in the local registry, we can sometimes find multiple object definitions in other registries. Another problem exists with the incorrect use of sets to describe the neighbors of an AS. Sometimes ASes, instead of creating a separate as-set to describe their neighbors, they use the set that describes what they import from these neighbors. This way we have an artificially large number of neighbors. We find and correct these cases.

After we build the database, the first thing we need to check is whether the ASes maintain the hierarchy of the sets. For example, consider the case that an AS constructs a set, which includes its customers and its local AS. If the provider of that AS includes the set in its own customer related set, then it maintains the hierarchy. If on the other hand it uses the members of the original set, and not the actual set, then that AS breaks the hierarchy. Our ability to analyze the policy is minimized when ASes don't maintain the hierarchy, especially for medium to large ASes. Intuitively, the number of nodes in the tree, versus the depth of the tree will give us an indication whether the ASes maintain the hierarchy. Recall that we model the sets using trees. In figure 8, we plot this graph for the as_sets and in 9 for the route_sets. From these figures, we observe that most of the as_sets seem to maintain the hierarchy, something that is not that evident in the route_set case.

The biggest flat as_set is the *AS-Sprint-Origins*, registered in the level3 registry, that has $7,310$ ASes. The overall largest as_set is the *AS-SeabonePeerEU* with a depth of 16 and an incredible large number of members, $96,166$. The set contains $10,053$ unique ASes. For the route_sets, the biggest one is the *RS-Level3-AS3356-Transit*, which has $68,538$ IP prefixes, and depth equal to 1. It is worth noting that the biggest flat sets for both the as_set and the route_set, are related with the Level3 ISP.

B. Phase Two: Inferring Policy

The next phase is to analyze the policy, and try to infer the actual business relations among the Autonomous Systems.

[7]Some of the sets missing are clearly simple spelling mistakes that the network administrator made. It seems that a large number of networks use the RPSL language only for registering a simplified version of their policy, and not for the actual configuration of their routers.

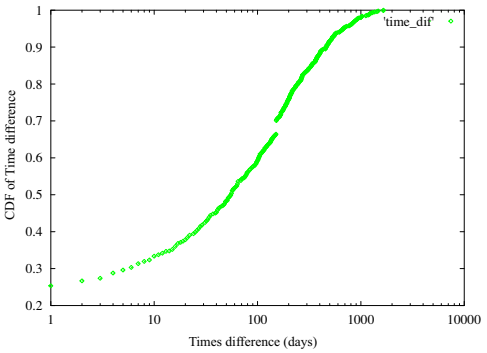

Fig. 12. CDF of the time difference between the last update time of an Autonomous System and the set it uses.

As we showed in the previous section, this is far from trivial. Different ASes have different uses of the IRR registries. Some ASes register very detailed policy, probably the one they use to configure their routers, other register a simplified policy, so that there exists a record in the IRR. Additionally, some ASes register only their customers, while others only their peers and providers. Finally, ASes don't always have their policy in a consistent state, their sets might be updated, but not their aut-num objects and vice versa.

Link-Level Transformation: Using our tool, we find that for 90% of the ASes, we can convert their policy to an equivalent link-level policy with no errors, i.e., for all export objects we can find an import object. In figure 10, we plot the degree of the Autonomous Systems in decreasing order, which have no errors, this corresponds to the 'perfect' line. We do the same for the Autonomous Systems that have errors, the 'errors' line. We observe that the higher the degree the more difficult it is to do the conversion. The largest AS that we can fully convert has 297 neighbors. In figure 11, we plot the number of missing objects versus the total number of objects in the policy. As we can see, there exist wide variations among the Autonomous Systems that can not be fully converted. On the positive side, there exist a number of medium to large Autonomous Systems that only miss a small number of objects.

Why we can not convert all the ASes? There can be three possible causes to this problem. First, the policy as

TABLE III
QUALITATIVE DIFFERENCE BETWEEN IRR AND ROUTEVIEWS

	All Links		Common Links	
Policy	**IRR**	**Gao**	**IRR**	**Gao**
Provider→Customer	21.3%	45.6%	33.3%	39.7%
Customer→Provider	21.2%	45.6%	42.7%	48.4%
Peer→Peer	56.3%	7.6%	22.3%	10.6%
Sibling→Sibling	1.1%	1%	1.6%	1.1%
Total Links	127,498	71,080	21,492	21,492

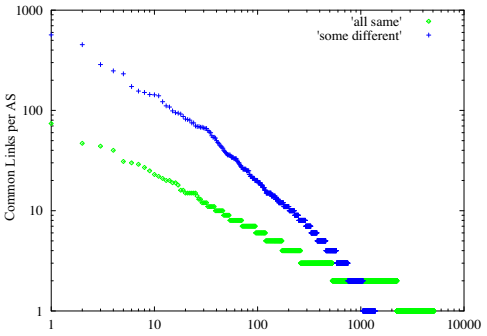

Fig. 13. Rank of the number of links per AS where either all links have the same type, or some have different.

registered might contain errors, such as not registering all the neighbors. Second, the policy can be more complex than the one our model tries to capture. Third, we have the case of policy inconsistency. For example, a set is updated more frequently than the Autonomous System that uses it. In order to further understand the reason, we will try to analyze the time difference between the Autonomous System and the sets it uses. In figure 12, we plot the CDF of the time difference between the Autonomous System and the sets it uses, for all the sets that we have errors. To determine the date the set was last updated, we find the last update of any of the sets that are included in the set, and the AS constructs them. For approximately 50% of the errors, we find that the difference is more than 2 months. This result makes us confident, that the reason we fail to convert to the link level policy is that the policy is not consistent.

Policy must be symmetric: We find that 650 more ASes fail on this test. In total, 82.8% of the ASes pass the first two tests. The major reason for asymmetry is the gap between the time the set and the AS was last updated. For example, consider the case where an AS changed providers, and the old provider changed only its set and not the Autonomous System policy. We can convert the policy to the link level with no errors, but the result is not correct, since the policy will be asymmetric. For example, the old provider will export its providers and peers to the customer, but the set it exports to the peers and providers will not contain the set of the customer AS, and so the policy on the link level will be asymmetric.

Inferring business relations: Based on the link level policy, we can find the business relations as described in the previous section. In Table III, we have a brief overview of our results. The all links column for the IRR, is when we take all the links that are registered in IRR, while the all links column for the Gao is all the links found in Routeviews and inferred their type using Gao's algorithm [21]. There exists both a quantitative and qualitative difference between these two datasets, at least as we have computed them. We find significantly more peer to peer links, compared to the Routeviews dataset.

C. Validation of IRR and our tool

In this phase, we try to assess the correctness of our tool, and the quality of information that exists in IRR. Using the database, we can build a network model at the BGP level. The model has all the necessary information to compare the IRR policies against actual BGP routing. We can check for origin misconfigurations by comparing the Autonomous System that originates the route in BGP against the registered origin in IRR. We consider a route announcement to be valid, if either the origin in both BGP and IRR is the same, or there exist a direct link between them and the type of relation is provider to customer or customer to provider. In order to catch export misconfigurations, i.e., BGP paths that contradict the intended policy, we will use the notion of the valley free paths [21]. We consider the provider to customer link to be down hill, the customer to provider link to be uphill, and the peers to be on the same level. For example, we can not have a path that goes from a provider to customer link to a peer to peer link. This path is not valley free.

Route Origin verification. There exist 135,398 unique routes in the Routeviews routing table. For 80% of them, we can find a corresponding record in IRR that can be used to verify the origin of the route. More specifically, 57% of the routes have a corresponding route in IRR with the same origin. For 23% of the routes, we have a less specific route, that contains the route found in BGP, and either has the same origin, or it is registered by a direct neighbor of the AS, which most of the times has a provider to customer type of relation. For 12% of the routes, there does not exist any route that can be used to compare the origin in IRR versus the origin found in BGP. An additional 8%, has a corresponding less specific route, but the origin can not be verified.

Links missing from IRR: We check whether all links in Routeviews can be found in IRR also. We find that only 38% of the ASes pass this additional test. As we mentioned, this can happen either because these ASes don't register all links, or because their Autonomous System policy is not fresh.

Business Relation verification: In order to evaluate our inference algorithm, we use the inference algorithm of Gao [21] on the Routeviews BGP table. Note that Gao's algorithm is a heuristic and may not always produce correct results. Its accuracy is reported around 96%.

There exist 21,492 common links in IRR and in Routeviews. We find that for 83% of the links, the type in our and Gao's approach is the same. Additionally, for 76% of the Autonomous Systems, all common links have the same type.

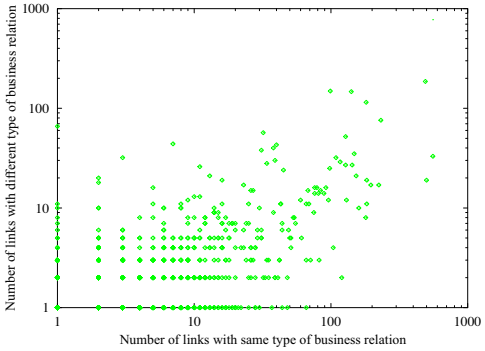

Fig. 14. Number of different versus same type of business relations for an AS.

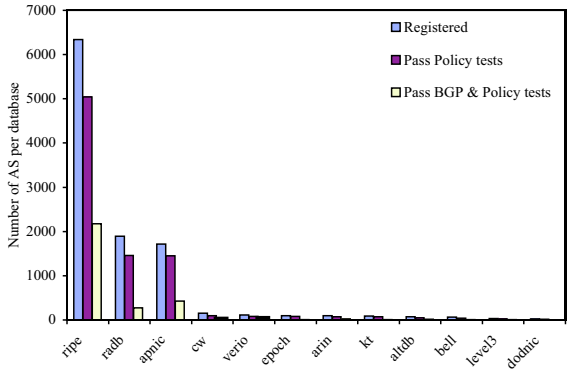

Fig. 15. Number of Autonomous Systems during our processing.

We should mention, that for an extra 10% of the links, we don't have a conflict in the actual paths in Routeviews, in the sense that these links don't cause a path to have a valley. We can consider that at least some of them are correct.

We want to examine where the two approaches differ. In figure 13, we plot the degrees in decreasing order, for the case where all common links have the same type as in Gao's, the 'all same' line. Additionally, in the 'some different' line, we have the degrees of the Autonomous Systems that only some of the links have the same type. Clearly, from the figure we can see that the bigger the ASes the less probable it is to have all the links with the same type. We have tried the case where we only pick ASes that have all links present and have correct policy, with a better accuracy, but in that case we have a limited number of links to compare.

In figure 14, we plot for each AS the number of links with a different type in our and in Gao's approach versus the number of links with the same type. There might be two reasons for the difference between our algorithm and that of Gao's. The first reason is that either our tool, or Gao's algorithm fail to correctly infer the type. The second is that the policy as registered is not correct. Studying the registered policies manually, we saw that often the fault is on how the policy is registered. The registered policy does not fully describe the actual accuracy of the IRR business relation.

Comparison of the registries: In figure 15, we plot the number of Autonomous Systems per registry that pass a given number of tests for correctness. The first box corresponds to the number of ASes registering their policy, the next box is the number of Autonomous Systems that pass all the policy tests, i.e., they can be fully converted to the link level and their policy is symmetric. In the last box we apply the additional test of comparing IRR with BGP routing. In order for an Autonomous System to pass this test it must first pass the policy tests, then register all the links found in Routeviews, and additionally, all links must have the same type of business relations. We see that RIPE is by far the most accurate registry, something that reinforces our belief that the difference in the inferred business relations is due to poor registered policy.

V. DISCUSSION: BGP ROBUSTNESS

Our tool is the interface between the IRR/RPSL language at the configuration plane, and the Internet routing paths at the operational plane. So far, we use our tool to process and validate IRR. Our tool discards inconclusive or erroneous information (first use of the tool). The question that arises is whether we can ever have an accurate IRR. This can happen either by compliance, or it can happen locally in an incremental way as neighbors collaborate and conform with mutual exchange of information.

In either of the three cases, starting from an accurate database, we can reverse the process, and we can validate routing updates using IRR. This can be used as an auxiliary consulting tool to raise flags, when something unusual appears. We consider this to be really important, since routing updates involve large amount of data, and it is impossible to manually check for errors. We have already developed the additional functionality. We have applied it to the updates collected from the Amsterdam Internet Exchange(AMS-IX) [32], and the results looks very promising. A tool that would check routing updates will receive the actual updates from a BGP speaking router. The processing will be done offline, in this case we can not prevent the errors, but we can identify them quickly and notify the administrator.

RIPE has developed a prototype, myAS [14], for exactly this purpose. It allows administrators to manually register the routes they want to safeguard, and the upstream providers of an AS. Our tool supersedes this current prototype, since we can use the actual RPSL policy of an AS. We believe that a tool that would automatically check for abnormal routing behavior, for all the policies stored in the registry, is something that would greatly improve routing robustness in the Internet. First, it would be an extra motive for AS administrators to use RPSL and the corresponding tools to configure their routers. This would result in fewer misconfigurations, since the manual configuration process, is much more error prone. Second, it would allow for misconfigurations to be quickly identified.

VI. CONCLUSIONS

We develop a methodology and tool for interfacing and cross-comparing the two major sources of BGP policy information: IRR at the configuration plane and BGP routing tables at the operation plane. On the one hand, the RPSL language is complex and obscure, while the BGP information is the end result of the policy and thus needs to be reverse engineered. Our tool bridges the gap between the two planes by providing novel capabilities.

As a proof of concept, we use our tool to obtain the following results.

- We quantify the quality of the current Internet Routing Registries. We find that 28% of the ASes have both a consistent policy and are consistent with BGP routing tables. Note though that almost all are from one only registry RIPE.
- We identify commons mistakes and problems in IRR registries. We discuss ways to overcome them so that IRR can be used to automate the management and safety of the Internet routing.

Through this analysis, we get strong evidence of the effectiveness of our methodology and tool. The 83% of the inferred policy is validated with external sources of information. The accuracy of our tool is surprisingly high considering that it has to detect and discard erroneous information.

Our ambition is to establish our tool as a foundation and inspiration for two complementary goals. First, we would like to draw the interest of the experts to develop efficient RPSL-based tools. Second, we would like to motivate practitioners and the related authorities to maintain and use more the IRRs. We think that one of the ways to succeed this is by establishing the practical potential of IRR. We view our tool to be a promising first step in this direction.

REFERENCES

[1] Y. Rekhter and T. Li, "A Border Gateway Protocol 4 (BGP-4)," 1995, RFC 1771.

[2] T. Griffin, F. Shepherd, and G. Wilfong, "The stable paths problem and Interdomain routing," *IEEE/ACM Transactions on Networking*, 2002.

[3] T. Griffin and G. Wilfong, "On the correctness of ibgp configuration," *ACM Sigcomm*, 2002.

[4] O. Maennel and A. Feldmann, "Realistic BGP traffic for test labs," *ACM Sigcomm*, 2002.

[5] R. Govindan, C. Alaettinoglu, K.Varadhan, and D.Estrin, "An architecture for stable, analyzable Internet routing," *IEEE Network Magazine*, 1999.

[6] "Internet Routing Registries," http://www.irr.net/.

[7] Alaettinoglu, C. Villamizar, E. Gerich, D. Kessens, D. Meyer, T.Bates, D. Karrenberg, and M.Terpstra, "Routing Policy Specification Language(RPSL)," RFC2622.

[8] D.Meyer, J.Schmitz, C.Orange, M. Prior, and C. Alaettinoglu, "Using RPSL in practice," RFC2650.

[9] C. Labovitz, A. Ahuja, A. Abose, and F. Jahanian, "An experimental study of delayed internet routing convergence," *ACM Sigcomm*, 2000.

[10] J. Cowie, A. Ogielski, B. Premore, and Y. Yuan, "Global routing instabilities triggered by code red ii and nimda worms attacks," Extended Technical Report, http://www.renesys.com/projects/bgp_instability/.

[11] R. Mahajan, D. Wetherall, and T. Anderson, "Understanding BGP misconfigurations," *ACM Sigcomm*, 2002.

[12] S.Murphy, "BGP security vulnerabilities analysis," Internet Research Task Force, 2002.

[13] "Gradus tool," http://gradus.renesys.com/.

[14] "MyAS Prototype," RIPE, http://www.ris.ripe.net/myas/.

[15] C. Lynn, J. Mikkelson, and K. Seo, "Secure bgp (S-BGP)," Internet-Draft, 2001.

[16] G.Goodell, W.Aiello, T. Griffin, J. Ioannidis, P. McDaniel, and A. Rubin, "Working around BGP: An incremental approach to improving security and accuracy of Interdomain routing," *NDSS*, 2003.

[17] N. James, "Extensions to BGP to support secure origin BGP (sobgp)," Internet Draft, 2002.

[18] N. Feamster and H. Balakrishnan, "Towards a logic for wide-area Internet routing," *Proc. ACM SIGCOMM Workshop on Future Directions in Network Architecture*, 2003.

[19] Y. Rekhter, B. Moskowitz, D. Karrenberg, G. de Groot, and E. Lear, "Address allocation for private internets," RFC-1918.

[20] "Internet Protocol V4 Address Space," http://www.iana.org/assignments/ipv4-address-space.

[21] L. Gao, "On inferring automonous system relationships in the Internet," *IEEE/ACM Transactions on Networking*, vol. 9, pp. 733–745, December 2001.

[22] G. Battista, M. Patrignani, and M. Pizzonia, "Computing the types of the relationships between autonomous systems," *IEEE Infocom*, 2003.

[23] L.Subramanian, S.Agarwal, J.Rexford, and R.Katz, "Characterizing the Internet hierarchy from multiple vantage points," *IEEE Infocom*, 2002.

[24] "Internet Routing Registry Toolset Project," http://www.ripe.net/ripencc/pub-services/db/irrtoolset/index.html.

[25] Q. Chen, H. Chang, R. Govindan, S. Jamin, S. J. Shenker, and W. Willinger, "The origin of power laws in Internet topologies revisited," *infocom*, 2002.

[26] A.Feldmann and J. Rexford, "IP network configuration for intradomain traffic engineering," *IEEE Network Magazine*, 2001.

[27] K. Poulsen, "Cracking down on cyberspace land grabs," http://www.securityfocus.com/news/5654.

[28] T. Griffin, A. D. Jaggard, and V. Ramachandran, "Design principles of policy languages for path vector protocols," *ACM Sigcomm*, 2003.

[29] T. Cormen, R. Rivest, and C. Leiserson, *Introduction to Algorithms*. MIT Press Inc., 1989.

[30] "Radb," www.radb.net.

[31] U. of Oregon Route Views Project, "Online data and reports," http://www.routeviews.org/.

[32] "Routing information service(ris)," www.ris.ripe.net.