Improving Dependability of Network Configuration through Policy Classification

Sihyung Lee Tina Wong Hyong S. Kim
Carnegie Mellon University
{sihyunglee, tinawong, hskim}@cmu.edu

Abstract

As a network evolves over time, multiple operators modify its configuration, without fully considering what has previously been done. Similar policies are defined more than once, and policies that become obsolete after a transition are left in the configuration. As a result, the network configuration becomes complicated and disorganized, escalating maintenance costs and operator faults. We present a method called NetPiler, which groups common policies by discovering a set of shared features and which uses the groupings for the configuration instead of using each individual policy. Such an approach removes redundancies and simplifies the configuration while preserving the intended behavior of the configuration. We apply NetPiler to the routing policy configurations from four different networks, and reduce more than 50% of BGP communities and the related commands. In addition, we show that the reduced community definitions are sufficient to satisfy changes as the network evolves over nearly two years.

1. Introduction

Network configuration is a low-level, device-specific task. To configure a network, one needs to configure each device in the network separately. There can be hundreds of devices, thus hundreds of configuration files, each with thousands of commands. Often, multiple files need to be modified to make a relatively minor change in the network. This requires the careful attention of operators since a change in one file can potentially affect other devices or even the whole network. These dependencies are spread across files of multiple devices, even in a small-sized network.

As a network evolves, its configurations become difficult to understand and to debug. Patches are sometimes put into configuration files to temporarily deal with a problem, and they are forgotten and left in place after the problem is handled. Old configurations often remain to ensure the network operation will work until the transition is complete. Configurations are edited by multiple operators with different backgrounds and working styles. In addition, networks are often merged into a single network, complicating the combined configurations. Also, because of the low-level nature of configuration commands, the same high-level goal can be achieved in various ways in configurations. In other words, both technical and non-technical issues can degrade the quality of a network's configuration over time.

As a result, companies spend more resources on the daily management and operations of their networks than on new IT services. In fact, one study has found that 80% of IT budgets in enterprise networks are used just to maintain the current operating environments [1]. Scheduled maintenance and upgrades can account for more than 30% of network outages in Tier-1 ISPs [2]. Operator errors are common and can account for more than 50% of failures in computer systems and networks [3][4].

Our system, called NetPiler, transforms the network configuration into a more manageable configuration. We define a configuration as manageable if the configuration is short, and if it can be extended over time according to the changes of the network, with few modifications. In this process of transformation, NetPiler extracts the underlying functions and dependencies from the network configuration and puts them into a concise and system-independent format by reducing any redundancy (Section 3.2). From this format, NetPiler generates a new configuration that takes into account complex inter-device and intra-device dependencies. We apply this technique to inter-domain routing policy configurations in order to demonstrate NetPiler (Section 4). We evaluate the algorithm on four production networks – two national providers and two regional providers (Section 5). We

are able to reduce up to 70% of the respective commands. We also go over a few reduction types and show that such simplification does improve the manageability of the configuration (Section 5.3). Finally, we present a few ways to improve the algorithm (Section 6).

2. Related Work

There has been a significant amount of work done to help simplify network management. To the best of our knowledge, this paper is the first to consider reducing a network configuration so as to increase its manageability. [5][6] propose high-level configuration languages for specific parts of a network configuration. NetPiler finds unique clusters of elements that share certain properties (or implement common functions) in a network configuration. Description by element groups can simplify the network configuration, independent of the description language. Others have proposed new management architectures. The 4D architecture [7] has a central decision plane, and CONMan [8] exposes a simple and consistent interface to the management plane. Even with these architectures, we believe that our method of transforming a configuration into a simpler form would make the configuration easier to maintain. [9][10][11] identify potential errors in configurations by comparing them to a list of predefined rules. Although these tools are effective for the detection of particular types of errors, their use is tedious because the operators must list the possible errors and customize the tools to the changes of the network. Our tool is more proactive and is compatible with these approaches. We remove the complexity and redundancies in the configuration that can increase maintenance costs and thus operator mistakes.

The goal of NetPiler differs from those of other optimization techniques found in VLSI CAD [12] and firewalls [13]. NetPiler transforms a configuration in order to increase the manageability and readability of the configuration for human operators. In contrast, the other optimization techniques are intended to speed up a program or to reduce the complexity of the compiled code, according to quantitative metrics. A manageable configuration for one aspect in the configuration may not be manageable for another aspect, and the generation of a manageable configuration requires domain knowledge about that aspect. Therefore, we do not attempt to generalize the method in NetPiler for all aspects. We show a way to enhance the general model for the inter-domain routing policy aspect (Section 4).

3. NetPiler

We first present an overview of NetPiler and show how we simplify a configuration for inter-domain routing policies and BGP communities in Section 3.1. We describe the details of NetPiler in Section 3.2 and its applications in Section 3.3.

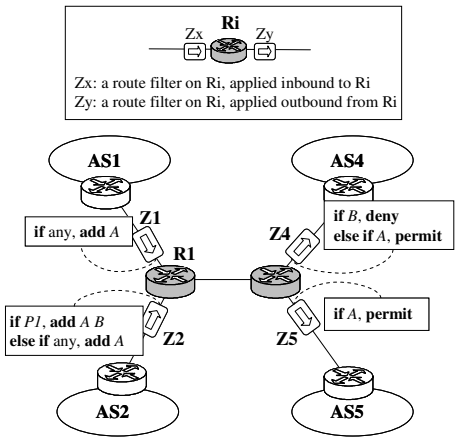
3.1. Overview

We perform the following steps to transform a network configuration into another form. We first select the element in the configuration that is subject to the transformation. The element can be the ones that can be grouped, ranging from routes that can be grouped by routing policies to packets that can be grouped by firewall or QoS policies. We then parse the configuration with regard to the element and construct a graph model. The model is a bipartite graph with two partite sets, the set of instances I and the set of their properties P. An instance $i \in I$ is joined by an edge with a property $p \in P$ iff i has p. For example, when we consider firewall policies, i refers to a certain collection of packets (e.g., packets from subnet 1.1.1.0/24), and its property p refers to the actions associated with the packets as well as the locations where the actions take place (e.g., sample and count the packets at router R). We use a graph model instead of simple sets since each instance can have multiple properties such that some of the properties are properties of other instances as well. A graph is easier and more natural to represent the overlapping nature of the relationships. From the model, we identify distinct groups of instances that share common properties. Group A is comprised of a set of properties P_A that characterize the group, and a set of instances I_A, each of which has all the properties in P_A. For example, we may identify a group of packets i_1, i_2, i_3 that are disallowed into *AS1*. We may identify another group i_4, i_5, i_6 that are allowed into *AS1*. This latter group of packets is tagged with IP precedence value 5 when forwarded to an external network *AS1*. If we denote the two groups A and B, I_A={i_1, i_2, i_3}, P_A={*discard at AS1*}, I_B={i_4, i_5, i_6}, and P_B={*permit, set IP precedence 5 at AS1*}. The two instance-property sets show two distinct policies associated with packets. Finally, we generate a new configuration that uses the groups in the specification.

Before we go into more detail, we start with a fictional scenario to illustrate what the scheme can do. The scenario includes routing policy configurations using the BGP community. We first present the background of inter-domain routing policies as well as the BGP community, and then the scenario.

3.1.1. Inter-domain Routing and BGP Communities. The BGP is a *de facto* standard inter-domain routing protocol. BGP route advertisement is selective in that only a subset of routes received from an AS is distributed to other ASes. This is done mainly to implement a business relationship or to engineer traffic between ASes [14]. The selection of routes works by applying a route filter to the BGP session to/from the AS. A route filter has a structure similar to the "if-then-else" chain in programming languages. It has a set of conditions followed by actions. The conditions and actions can be comprised of many different attributes in a BGP route such as AS-path and destination prefix. Among the attributes, the BGP community is one of the most widely used.

A BGP community refers to a group of routes that share certain properties, and thus the same action is applied to the community. A community is encoded as a 32-bit field. A community influences the selection of routes by having its 32-bit string tagged to the set of route advertisements that belong to the community. If the 32-bit string matches the condition of a route filter, the required action is performed. A community implements a routing policy, which is in general described by a 3-tuple, (description of a set of routes, actions to be taken on the routes, a set of local/remote locations for the actions). For example, (All the prefixes received from AS1, re-advertise, outbound session to AS5) means that we want all the routes received from the inbound BGP session with AS1 to be re-announced to AS5. To implement the policy using a community, a community *A* is added to the routes by a route filter that is applied to the inbound direction of BGP sessions with AS1. The route filter has the condition "`if any prefixes`", and the action "**`add`** *`A`*". Another router filter in the outbound direction with AS5 will announce the routes to AS5 by the condition "`if there exists community A`", and the action "**`then`** `permit`". Fig. 1(a) illustrates an example implementation of the same policy. The shaded routers are in our administrative domain, and the other routers are in external networks. A line between two routers denotes that there exists a BGP session between the routers. A rounded rectangle represents a route filter. The arrow within the filter indicates the direction where the filter is applied. The actual content of the filter (i.e. an if-then-else chain) is connected with a dashed line. For example, router filter *Z1* is applied to the routes advertised from AS1 towards *R1*. The filter adds community *A* to all the routes from AS1. These routes match the condition of *Z5* and are re-announced to AS5. Every if-then-else chain has an implicit deny action at the end. Thus, all the other routes are disallowed by the default deny action. There is a variety of community usages, and more details can be found in [15].



(a) The implementation of communities *A* and *B* in (b). ASes 3 and 6 are not shown for simplicity.

Comm.	*I*: set of routes	*P*: common actions & locations
A	any from {1,2,3}	Advertise to {4,5,6}
B	prefixes *P1* from {2} prefixes *P2* from {3}	Do not advertise to {4}

(b) Initial configuration

Comm.	*I*: set of routes	*P*: common actions & locations
A	any from {1,2,3,13}	Advertise to {4,5,6}
B	prefixes *P1* from {2} prefixes *P2* from {3}	Do not advertise to {4}
C	any from {7,8,9,13}	Advertise to {4,5,6}
D	any from {1,2,3,7,8,9}	Advertise to {4,5,6}
E	any from {1,2,3,7,8,9,13}	Advertise to {10,11,12}
F	prefixes *P1* from {2} prefixes *P2* from {3}	Do not advertise to {5,6}

(c) Configuration after network evolution

Comm.	*I*: set of routes	*P*: common actions & Locations
A'	any from {1,2,3,7,8,9,13}	Advertise to {4,5,6,10,11,12}
B'	prefixes *P1* from {2} prefixes *P2* from {3}	Do not advertise to {4,5,6}

(d) Simplified configuration after applying NetPiler

Figure 1. An example scenario on inter-domain routing and BGP community. The italicized letters *A* through *F* represent BGP communities. *P1* and *P2* are particular sets of destination prefixes. The second column (i.e. the set of routes) represents instance sets, whereas the third column (i.e. common actions & locations) represents property sets.

3.1.2. Overview Example. In the scenario, we show how a network configuration becomes convoluted as communities are added and replaced ad hoc, and how we reduce the complexity. To better illustrate the routing policies in the network, we use the table as shown in Fig. 1(b). Each row represents a routing policy group implemented by a community. The letter on the first column is the community that implements the policy. The second and third columns represent instance sets and the respective property sets. For example, Community *A* implements the policy group, "all routes from ASes {1,2,3} are re-advertised to ASes {4,5,6}." This group has three members in I_A={any prefixes from ASes 1,2,3} and is characterized by P_A={advertise to ASes 4,5,6}. *P1* and *P2* in community *B* represent certain collections of prefixes from ASes 2 and 3, respectively. When there are multiple rules for the same route, the most specific rule precedes the other rules. For example, regarding the advertisement pattern to AS4, the second policy applies to prefixes *P1* from AS2, whereas the first policy applies to the rest of the routes from AS2. Thus, all routes from AS2 are advertised to AS4 except the prefixes *P1*. The actual implementation of the two policies is shown in Fig. 1(a). For simplicity, we omit the sessions with ASes {3,6}. Route filter *Z2* tags prefixes *P1* with community *B*. Therefore, the prefixes are filtered out by *Z4* and are not allowed into AS4.

Fig. 1(b) shows the initial configuration in the scenario, and Fig. 1(c) shows the configuration after the network went through changes. We show only the two configurations in the evolution because of space limitations.

- Initially, there are six neighboring networks, ASes 1 through 6. One community *A* is used to re-advertise routes from ASes {1,2,3} to ASes {4,5,6}.
- It is decided that IP prefix *P1* from AS 2 and prefix *P2* from AS 3 are not re-advertised to AS 4. Community *B* is set on the IP prefixes and matched by a new outbound statement towards AS 4 to deny the IP prefixes. This situation corresponds to Fig. 1(b).
- The network establishes a peering relationship with three new ASes {7,8,9}. Community *C* is defined to re-advertise routes from ASes {7,8,9} to ASes {4,5,6}.
- There is a merger of networks, and the operators decide to replace communities *A* and *C* with a new community *D*. In the procedure, *A*s and *Cs* remain in the configuration in order to prevent any malfunction while the migration is incomplete.
- Three new neighbors, ASes {10,11,12} are added, and a new community *E* is defined so that the three new neighbors receive routes advertised from ASes {1,2,3,7,8,9}.
- A new neighbor session to AS 13 is negotiated by a new operator. Without being aware of community *D*, the operator applies two old communities *A* and *C*. Community *E* is also applied.
- IP prefixes *P1* and *P2* from ASes 2 and 3, respectively, are no longer re-advertised to ASes 5 and 6 by a new community *F*.

The configuration after the network extension (as shown in Fig.1(c)) is much more complex than its initial form with two communities *A* and *B*. There are six communities, each of which forms a certain routing policy group. NetPiler can re-cluster the policies into two distinct groups, and the result is shown in Fig. 1(d). Each of these two groups can be implemented by a single community. Note that this simplified configuration is functionally equivalent to the intended policies. In other words, any route received from any neighbor will take the same action at any location as in Fig.1(c). As illustrated by the example, our aim is to make the configuration more manageable by combining similar groups, eliminating unused groups, and better expressing the used groups.

3.2. Instance-Property Model and Decomposition

An element in a network configuration can be described with a set of properties associated with it. Our model captures such relationships between the element's instances and its properties in order to identify groups of instances sharing common properties and to simplify the configuration through grouping. We call this model an *instance-property model*. In the model, a relation of an instance *i* having a property *p* is represented by two vertices *i* and *p* having an edge between them. In other words, our model is a bipartite graph with partite sets *I*, the set of instances, and *P*, the set of properties associated with the instances such that instance $i \in I$ is adjacent to a property $p \in P$ iff *p* characterizes *i*. Fig. 2(a) shows an instance-property model *G* with five instances and seven properties. Instance i_1 has 4 properties {p_1, p_3, p_4, p_7} and thus is incident with 4 edges that are joined with {p_1, p_3, p_4, p_7}.

It is clear that an instance-property model can be described by listing each relation *(i,p)* represented by an edge. However, our goal is not to separate each single edge. We partition the edges into sets, such that each set represents a distinct group of instances that share certain properties as a unit. We call such a partition a *decomposition* of the model. Grouping

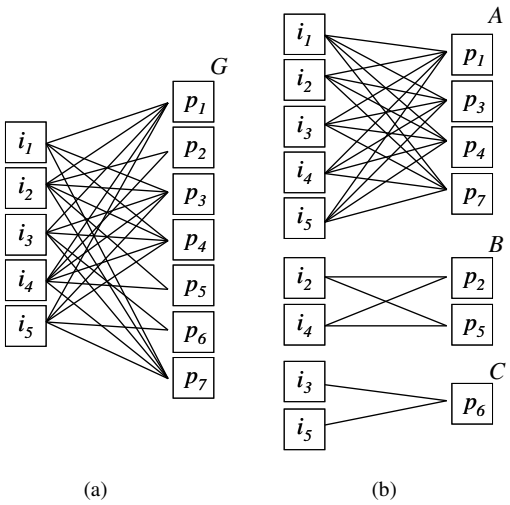


Figure 2. A decomposition of an instance-property model G (as shown in (a)) into complete bipartite subgraphs A, B, and C (as shown in (b)).

similar objects and representing the objects by group improve the manageability. We define a group as follows. Group A is a nonempty set of properties P_A together with a set of instances $I_A = \{i | i \in I_G, P_i = P_A\}$. $P_i = \{p | p \in P_G, (i,p) \in E(G)\}$. G denotes the instance-property model and $E(G)$ its edge set. Since in A, every instance in one partite set I_A is adjacent to every property in the other partite set P_A, a group is equivalent to a *complete* bipartite graph. Thus, partitioning G into groups is the same as decomposing G into complete bipartite subgraphs. Fig. 2(b) presents a decomposition of G in Fig. 2(a) into 3 complete bipartite graphs (groups), A, B, and C. If the instances are routes, then A, B, and C can represent "routes advertised from ASes {1,2,3,4,5}", "routes received at router R1", and "routes received at router R2". Each group may have different properties such as "advertise to AS 6", "advertise to AS 7", and "prepend the AS n times in the AS-PATH attribute when the routes are advertised to AS 7". Note that i_2 belongs to both A and B. Such a membership is a single new group that inherits the properties from A with the addition of the properties from B. The decomposition of G is function-preserving: we do not add or delete any edges in G, and thus the intended behavior of the configuration does not change although its specification does.

Note that there are many ways to decompose G into groups. For example, G is also decomposable into three groups A', B', and C' with their I and P sets as follows: $I_{A'} = \{i_1\}$, $P_{A'} = \{p_1, p_3, p_4, p_7\}$, $I_{B'} = \{i_2, i_4\}$, $P_{B'} = \{p_1, p_2, p_3, p_4, p_5, p_7\}$, $I_{C'} = \{i_3, i_5\}$, and $P_{C'} = \{p_1, p_3, p_4, p_6, p_7\}$. Of all possible decompositions, we look for the decomposition where each group is manageable (i.e. an operator can reuse the groups to specify new instances or to modify existing instances with or without slight modification in the group definitions, and the meaning of the groups is consistent so that it is straightforward to grasp the meaning of the groups.).

A manageable decomposition for one type of element may not be manageable for another type of element. Thus, identification of a manageable decomposition requires domain knowledge about the instance. In Section 4, we suggest one method to find a manageable decomposition, especially for inter-domain routing policies and the BGP community.

3.3. Applications of NetPiler

In this section, we investigate which aspect of a network configuration can be simplified by NetPiler. There are cases where grouping is explicitly used with group ID. These cases include route tagging based on routing policies, packet marking/grouping based on QoS policies, and MPLS labeling based on destination prefixes/packet treatments. ACLs (Access Control List) in a network can also be grouped into distinct sets of policies. Since all routing/QoS/ACL policies are based on filters, which are essentially if-then-else chains, we can use the same technique as shown in Section 4 to identify instances and properties. The instance set I could be a set of routes/packets, and the property set P could be a set of actions on the routes/packets and locations of the actions.

The routing policies and ACLs comprise a major portion of the network configurations in the observed networks (i.e. up to 70% of a configuration file), and they are modified frequently, often within 10 days of the previous changes [16]. In particular, the networks rely heavily on BGP communities to tag routes and control announcements. Therefore, we chose to present the application of NetPiler in BGP communities. BGP communities are particularly troublesome in large carriers. There are hundreds of different communities, and tens of these communities are used in each command line. Network configuration using large number of communities is tedious, difficult to understand, and prone to human errors. We observe numerous errors related to BGP communities in the networks that we study. We believe the application in BGP communities would better illustrate the benefits of our method.

We are currently working on extending the applications. For example, interface configurations can be grouped into "external interface class", "interface

class facing neighbor N1", and "interface class facing neighbor N2". Such description is possible in JUNOS by using the `group` command [20].

4. Demonstration with Communities

4.1. Construction of Instance-Property Model

At a high level, we construct the instance-property model for routing policies that are implemented by communities. We then decompose the model into groups such that each group represents a distinct routing policy as a unit and therefore is assigned to a different community.

We identify an if-then-clause in a route filter as an instance. If we think of a community in terms of a group defined in Section 3.2, the members of the community (i.e. the instances of the community) are the routes tagged with the community. In a configuration, the routes are represented by sets of conditions in one or more route filters, possibly applied to different neighbors, such that each set is matched as a unit. One such set of conditions is equivalent to an if-then-clause. In Fig. 1(a), there are three if-then-clauses that represent instances of community *A*: i) all routes from AS1, ii) prefixes *P1* from AS2, and iii) the rest of the prefixes from AS2.

Similarly to instances, we identify an if-then-clause in a route filter as a property. In other words, each if-then-clause will become an instance as well as a property. The properties of the community are local/remote locations where the routes are matched. These locations are associated with the actions that take place on the routes. In a configuration, the local/remote locations and the actions are represented by if-then-clauses that match the community. In Fig. 1(a), there are two if-then-clauses that match community *A*, and they are applied outbound to AS4 and AS5.

The edges of the instance-property model, relationships between instances and properties, are identified as follows. There is an edge between one if-then-clause *i* and another if-then-clause *p* if the routes represented by *i* are matched by *p* via communities (i.e. if the communities attached by *i* match the condition in *p*). For example in Fig. 1(a), the routes received from AS1 have community *A* attached by the if-then-clause "`if any, add A`". These routes match the if-then-clause in filter *Z5*, "`if A, permit`". Therefore, the two if-then-clauses are joined by an edge. For an edge *(i,p)*, routes matched by *i* flow through *p* and the actions specified in *p* are taken on the routes. In the next section, we identify distinct policy groups that are represented by the dependencies among if-then-clauses, and we assign a community to each routing policy so that the community is used in its associated if-then-clauses.

4.2. Identifying Distinct Policies

Once an instance-property model is obtained, there are many ways to decompose the model. Naive decomposition may lead to groups that are difficult to reuse. Thus, we develop a condition for each group to be manageable. Although the condition is further refined, we focus on the essence in this section. The extensions are presented later in Section 6.

The condition is based on the observation that a routing policy described by a community generally involves a set of routes that require the same set of actions. For example, routes from all customers might be re-advertised to all the peers and providers. A few prefixes from some customers might be AS-prepended three times when re-advertised to other peers so that those routes are not preferred. Such different sets of routes are represented by instances in our model. Thus, in order to identify distinct sets of routes that cause certain actions in concert, we identify such sets of instances.

We formalize the algorithm in Fig. 3 and present an example in Fig. 4. In a policy model *G*, we go over each property p_y and identify the set of instances $I_{tmp}(y)$ that are adjacent to p_y. $I_{tmp}(y)$ represents the set of routes that match the condition of p_y and thus are subject to the same action as described in p_y. Among all such sets, we draw distinct sets, I_1 through I_N. These sets represent distinct sets of routes that take the same action. Each I_x has its counterpart P_x, $\{p_y: I_{tmp}(y)=I_x\}$. For each pair (I_x, P_x), all the edges between (I_x, P_x) belong to the same group and thus are assigned to the same community. In Fig. 4, there are two distinct I_x's that take the same actions as a unit, $I_1=\{i_1\}$ and $I_2=\{i_1, i_2, i_3\}$. The corresponding P_x's are $P_1=\{p_1\}$ and $P_2=\{p_2, p_3\}$. The two routing policy groups use community *A* and *B*, respectively. The edges in the original configuration (a) and the reproduced configuration (d) are the same, and thus the transformation is function-preserving. Note that each community (group) in the reproduced configuration has a consistent meaning. In fact, a community represents a "*come-from*" relationship: routes that come-from I_x take certain actions in P_x as a unit.

5. Evaluation

We implement and evaluate our algorithm for the communities on configurations from four different

i_x : x-th instance
p_y : y-th property
G : Policy model. $G_{x,y} = 1$ if $(i_x, p_y) \in E(G)$. Otherwise, $G_{x,y} = 0$.
N : Number of new communities
c_x : x-th new community
I_x : A set of instances that adds c_x
P_x : A set of properties that match c_x
$h(\ )$: Hash function associated with a hash table H. If $h(I_{tmp})=x > 0$, I_{tmp} is present in H, where $I_x = I_{tmp}$. Otherwise, $h(I_{tmp}) = 0$.

Empty H.
$N = 0$;
for each property p_y
    $I_{tmp} = \varphi$;
    for each instance i_x
        if $G_{x,y} = 1$ **then** $I_{tmp} = I_{tmp} \cup \{i_x\}$;
    if $h(I_{tmp}) = 0$ **then** { // create a new community
        $N = N + 1$; $h(I_{tmp}) = N$;
        $I_N = I_{tmp}$; $P_N = \{p_y\}$;
    } **else** $\{P_{h(I_{tmp})} = P_{h(I_{tmp})} \cup \{p_y\};\}$

Figure 3. Algorithm that identifies distinct policies based on the come-from relationship.

production networks. The evaluation is done in three steps. First, we assess the reduction in the configuration length. We use two complexity measures that are proven to have strong correlation with maintenance cost. Second, we examine the meanings of the policy groups before and after the transformation. For these two steps, we analyze a particular snapshot of each network between March and April 2006. Finally, we analyze monthly snapshots of network *1* and *2* for two years to see if communities generated by our algorithm for the first snapshot could be reused over time. As shown in Table 1:

- We reduce up to 90% of communities and 70% of community related commands. If we disregard communities that do not create any edges (Section 5.3), no reduction is possible for two networks either because there is a simple set of policies, their communities are well structured, or there have not been many changes.
- More than 70% of the communities are defined by the come-from policy. There are a few exceptions, and we address them in Section 6.
- Most new communities are shown to be reusable as the number of peering relationships grows by 25% over the two-year period.

We describe implementation/experimental details in Section 5.1 and the two complexity measures in Section 5.2. We then present the details of our results in Section 5.3.

5.1. Experimental Setup and Implementation

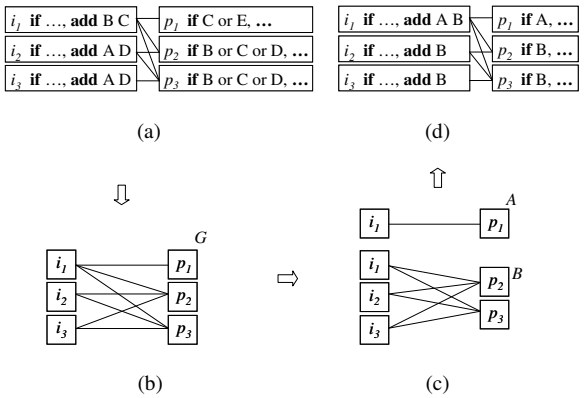


Figure 4. An example of routing policies (as shown in (a)), the corresponding instance-property model (as shown in (b)), decomposition by the come-from relationship (as shown in (c)), and the reproduced routing policies (as shown in (d)).

First, we focus on the simplification and restructuring of internal BGP communities within one administrative domain. We do not consider communities that are intended for use by external networks. However, this idea can be extended to multiple domains in the same way. In addition, predefined standard communities such as *no-export* and *no-advertise* are not subject to our simplification process.

Our implementation uses a configuration parser [9] developed for Cisco IOS and Juniper JUNOS commands. We parse routing policies related to communities and separate if-then-clauses into instances/properties in the format shown in Fig. 5. A property has a condition in Boolean logic since communities are matched based on Boolean operations (AND/OR/NOT). An instance has a list of communities attached by its corresponding if-clause. Although a community can be deleted as well, for simplicity we consider only the addition of communities. In the configurations from the four networks, we find that deletion of communities is rarely used, and it is only used to remove certain communities on routes received from/advertised to external networks. Therefore, deletion of such communities does not influence the operations of communities used within the administrative domain.

Fig. 5 shows an instance-property model representation for a configlet of Cisco IOS. There are two route filters, `from_dora` and `to_toto`. We also show their instance-property model. Instance i_1 and property p_1 represent `from_dora`, whereas i_2 and p_2 represent `to_toto`. The edge (i_1, p_2) indicates that routes redistributed through `from_dora` will match `to_toto`. Refer to our technical report [19] for details.

TABLE 1. SUMMARY OF ANALYSIS

Index	Num. communities		Num. LOC	
	Before	*After*	*Before*	*After*
1	293 (113)	8	9003 (8419)	2036
2	43 (4)	4	282 (184)	194
3	45 (14)	10	2756 (1443)	1409
4	11 (4)	4	227 (126)	126

Network {1, 2} are regional providers, and Network {3, 4} are national providers. The number of routers are (44, 6, 13, 11) and the number of distinct external peers are (133, 39, 414, 77). The numbers in parentheses represent the numbers excluding dangling communities (as shown in Section 5.3) that do not create any edge.

Note that an edge can also be created by other types of filters based on prefix or AS-path attributes. In that sense, this model can be extended to a layered model with edges that are made from communities, and which are elaborated by conditions involving prefixes, AS-path, and so forth. We do not consider these additional conditions in this paper. However, we preserve the edges made from communities when reassigning communities. Therefore, the edges will be correctly elaborated by other conditions that are not considered in this process, and the underlying routing policies are left intact.

5.2. Complexity Measures

We use two measures, the number of communities and the number of LOC (Lines of Commands). Multiple studies validate their correlation with the number of faults and development/maintenance time required [17][18]. The number of communities measures the total number of distinct communities that are used internally within a network. This is analogous to the vocabulary size [17], a software complexity measure that counts the number of unique operators and operands. It reflects the size of search space when writing or reading a command. As it becomes larger, the operator has to consider and compare more options to configure a community, and the configuration becomes a more complex task. The number of LOC is the sum of the number of individual communities used in conditional/action clauses. This also has its counterpart, which counts the number of individual commands in software. The more places an operator needs to configure, the more chance to make mistakes. This is especially true when we configure communities, each of which can have dependencies and can impact tens to hundreds of BGP sessions [16]. Furthermore, our previous work [9] finds a number of community-related errors, such as missing communities and using wrong communities in a network where each if-then-clause consists of more than five communities on average.

Configuration in Cisco IOS syntax:

```
01  neighbor 1.1.1.1 route-map from_dora in
02  neighbor 2.2.2.2 route-map to_toto out
03
04  route-map from_dora permit 10
05   match community LIST1
06   set community 1:200 1:300
07  !
08
09  route-map to_toto permit 10
10   match community LIST2
11   set community 3:500
12  !
13
14  ip community-list LIST1 deny 2:444
15  ip community-list LIST1 2:100
16  ip community-list LIST2 1:200 1:300
17  ip community-list LIST2 1:400
```

Instance-Property Model Representation:

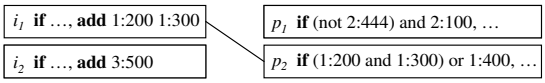


Figure 5. BGP configuration in instance-property model representation.

5.3. Results

Table 1 shows decreases in both the number of communities and the number of LOC. The decrease is noticeable in Network *1* since it had been expanding as more networks were added over the span of the two years we studied and thus had gone through many changes in the past. Note that some redundancies are by design, and operators can always keep certain original communities from being restructured. The operators can either exclude the original communities from the analysis, or accept only a subset of the new communities.

Each of the new communities either is equivalent to an original community or represents policies implemented by multiple communities in the original configuration. Some of the new communities implement business relationships among transits, peers, and customers, while others implement policies intended for traffic engineering. These are common relationships found in a network, and configurations concerning communities thus naturally can be reduced according to the unique units of these relationships in a network.

Dangling Communities. The majority of communities that are removed by our algorithm (180, 39, 31, and 7 communities from network 1, 2, 3, and 4, respectively) are either added in if-clauses but never matched anywhere, or matched but never added. We call these communities dangling communities since they refer to a certain group, but do not form any edges in the instance-property model. These communities are remains of old configurations when peering

relationships end. Others are defined by predicting later usage thus allowing operators to use the communities to deal with modification in peering relationships or unforeseen problems in the future. However, from our time-series analysis over a two-year period, we find that none of these communities had been modified for actual usage. These communities should be used only when they are needed. Lengthening the configuration with such communities might make the configuration harder to understand, maintain, and more prone to errors.

Subset Communities. A few communities are removed since their functions are subsumed by those of other communities. In other words, the edges created by each of the removed communities are a subset of the edges created by another community. In one network, particular routes are re-advertised to a peer based on the following matching condition.

if (*A* and *C1*) or (*A* and *C2*) or (*A* and *C3*) or …

Our algorithm detects that wherever A is attached, one of the *Ci*'s is attached as well and thus is able to simplify the condition as "**if** *A*."

There are two possible reasons why such communities exist: i) when communities are defined ad hoc, the dependencies created by communities and the policies implemented previously are not fully considered, or ii) communities that are replaced by others are not properly removed.

Combination of Communities. There are communities that can be combined although none of them are functionally subsumed by one another. Such communities either represent the same set of routes and match in different if-clauses, or involve different routes and match in the same if-clauses. For example, three communities are added by the same if-then clauses and thus represent the same set of routes. The communities are used so that the routes are not re-advertised to three different networks *1*, *2*, and *3*, respectively. Our algorithm combines the three communities as one by matching and adding a single community instead of the three. Such combining does not limit the flexibility of routing policies as long as we deal with the same set of routes. If we no longer need to prevent the routes from being advertised to network *2*, we can simply remove the single community from the corresponding if-clause.

Equivalent Communities. Each of the other new communities (3, 4, 7, and 4 communities from network 1, 2, 3, and 4, respectively) is equivalent to an original community. Although these communities do not contribute to the reduction, they do present an important implication as the combined communities. This implication is that the majority of routing policies comply with the come-from relationship. There are a few exceptions, which we deal with in Section 6.

Time-series Analysis. Finally, we perform an analysis on snapshots that cover a two-year period (Network *1* and *2*). The result is encouraging because it shows that configurations from a simple transformation can still be evolvable over time. During the period, the networks add and remove peering relationships periodically, and the overall number of relationships grows by roughly 25%. We find that the reduced set of communities is sufficient for this evolution. One or two communities are added and then deleted during the period to accommodate temporary peering relationships that require unique routing policies.

6. Discussion

In this section, we go over a few cases where the number of communities/LOC does not decrease when the new groupings reproduced by the come-from relationship disagree with the groupings in the original configurations. Since we believe that the original groupings could be more meaningful, we present methods that restructure the new groupings into the original groupings to improve the come-from relationship. More details can be found in our technical version of the paper [19].

Preference for Shorter Descriptions. A shorter description could be more intuitive than a longer one. For example, "All but routes from AS1 are to be advertised to customers." is more concise than "Routes from ASes {2,3,4,…,n} are to be advertised to customers." The come-from relationship produces the latter grouping while the original configuration uses the former. The latter requires a community *A* to be attached to the routes from each of the n-1 ASes {2,3,4,…n}. The community is matched by "`if A, permit`" when the routes are advertised to the customers. On the contrary, to implement the former grouping, we can use negation in the if-clause as in "`if (not A), permit`". This requires the community *A* to be added only to the routes from AS1 and thus reduces the LOC. Although the situation that we describe here is not common, when it happens, we observe a tendency towards using smaller *I* and *P* sets or fewer communities.

Finer Decomposition Based on Actions. We can further partition the policies resulting from the come-from relationship in order to make their meanings clearer. Assume that a set of prefixes *P1* learned from external peers is either dropped or receives a lower preference at two different remote route filters. The come-from relationship identifies the situation as one

single policy, "come-from *P1*," since the prefixes always receive the same action as a unit. However, we can divide the policy into two policies: i) "come-from *P1* to be dropped," and ii) "come-from *P1* to receive a lower preference." If the latter is used, one can easily extend our algorithm so that come-from based policies are further partitioned according to the corresponding actions.

7. Conclusion

We present NetPiler, a way to transform a network configuration into a simpler form, which is easier to read and update. NetPiler groups policies into a set of distinct policies, thus removing any duplicate specifications, and it combines specifications that are unnecessarily decomposed. We demonstrate NetPiler for routing policies in four production networks, especially the policies implemented by the BGP community attribute. We show that up to 90% of communities and up to 70% of community-related commands are reduced. We also run NetPiler for snapshots over two years and show that the reduced set of communities can be reused and are sufficient for this evolution.

The respective operators find NetPiler helpful for managing and understanding their network configurations. The strength of NetPiler is not only that it helps change the existing configurations, but it also represents the configurations in concise manners, thus paving a way to improve the readability of the configurations. NetPiler simplifies hundreds of policies into roughly ten policies, and the operators understand such a representation better than the original configurations. This representation also leads the operators to identify policies that are not intended or misconfigured. Thus, we believe that NetPiler can potentially reduce operator mistakes as well as maintenance costs, making the network more reliable and dependable. Finally, we hope to conduct user studies that will involve many operators of various skill levels to see if the resulting configuration files are more manageable.

8. References

[1] Z. Kerravala, "As the value of enterprise networks escalates, so does the need for configuraiton management," Enterprise Computing and Networking, Yankee Group, 2004.

[2] "Evaluating high availability mechanisms," Agilent Technologies White Paper, 2005.

[3] R. Mahajan, D. wetherall, and T. Anderson, "Understanding BGP misconfigurations," in *Proc. ACM SIGCOMM*, Aug. 2002.

[4] D. Oppenheimer, A. Ganapathi and D. Patterson, "Why do Internet services fail, and what can be done about it?" in *Proc. USITS*, 2003.

[5] C. Alaettinoglu, et al., Routing Policy Specification Language (RPSL), RFC-2622, 1999.

[6] T. Griffin, A. Jaggard, and V. Ramachandran, "Design principles of policy languages for path vector protocols," in *Proc. ACM SIGCOMM*, Aug. 2003.

[7] A. Greenberg, et al., "A clean slate 4D approach to network control and management," *ACM SIGCOMM Computer Communications Review*, vol. 35, no. 5, Oct. 2005.

[8] Hitesh Ballani and Paul Francis, "CONMan: A step towards network manageability," in *Proc. ACM SIGCOMM*, Aug 2007.

[9] F. Le, S. Lee, T. Wong, H. S. Kim, and D. Newcomb, "Minerals: Using data mining to detect router misconfigurations," in *Proc. ACM SIGCOMM Workshop on Mining Network Data*, Sep. 2006.

[10] N. Feamster and H. Balakrishnam, "Detecting BGP configuration faults with static analysis," in *Proc. NSDI*, May 2005.

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

[12] R. Brayton, A. Sangiovanni-Vincentelli, C. McMullen, and G. Hachtel, Logic Minimization Algorithms for VLSI Synthesis, New York: Kluwer Academic, 1984.

[13] A. Liu, E. Torng, C. Meiners, "Firewall compressor: an algorithm for minimizing firewall policies," in *Proc. IEEE Infocom*, Apr. 2008.

[14] M. Caesar and J. Rexford, "BGP routing policies in ISP networks," *IEEE Network Magazine, special issues on inter-domain routing,* Nov/Dec. 2005.

[15] O. Bonaventure and B. Quoitin, "Common utilizations of the BGP community attribute," Internet draft, draft-bonaventure-quoitin-bgp-communities-00.txt, work in progress, June 2003.

[16] S. Lee, T. Wong, and H. S. Kim, "To automate or not to automate: on the complexity of network configuration", in *Proc. IEEE ICC, May 2008.*

[17] H. Zuse, Software Complexity: Measures and Methods, Berlin: Walter de Gruyter, 1991.

[18] S. Alexandrov, "Reliability of complex services," unpublished.http://www.cs.rutgers.edu/~rmartin/teaching/spring06/cs553/papers/

[19] S. Lee, T. Wong, and Hyong S. Kim, "NetPiler: Reducing network configuration complexity through policy classification," CMU Technical Report, CMU-CyLab-07-009, 2007.

[20] JUNOS Configurations Guides. http://www.juniper.net/techpubs/software/junos/junos83/index.html