# NetPiler: Detection of Ineffective Router Configurations

Sihyung Lee, *Student Member, IEEE,* Tina Wong, and Hyong S. Kim

***Abstract*—Configuring a network is a tedious and error-prone task. In particular, configuring routing policies for a network is complex as it involves subtle dependencies in multiple routers across the network. Misconfigurations are common and certain misconfigurations can bring the Internet down. In 2005, a misconfigured router in AS 9121 blackholed traffic for tens of thousands of networks in the Internet. This paper describes NetPiler, a system that detects router misconfigurations. NetPiler consists of a routing policy configuration model and a misconfiguration detection algorithm. The model is applicable to routing policies configured on a single router as well as to network-wide configuration. Using the model, NetPiler detects configuration commands that do not influence the behavior of the network - we call these configurations ineffective commands. Although the ineffective commands could be benign, sometimes when the commands are mistakenly configured to be ineffective, they cause the network to misbehave deviating from the intended behavior. We have implemented NetPiler in approximately 128,000 lines of C++ code, and evaluated it on the configurations of four production networks. NetPiler discovers nearly a hundred ineffective commands. Some of these misconfigurations can result in loss of connectivity, access to protected networks, and financial implications by providing free transit services. We believe NetPiler can help networks to significantly reduce misconfigurations.**

***Index Terms*—Network abstraction, network configuration modeling, network management, static analysis.**

## I. Introduction

CONFIGURING a network is a low-level and 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. A change in one device can potentially affect other devices or even the whole network. Often, multiple devices need to be reconfigured to make a relatively minor change in the network. Network configuration files are complex due to subtle dependencies among them. These subtle dependencies exist in different parts of a single file and spread across files of multiple devices, even for a small sized network. For example, the network policy that states, "allow a set of packets to go from router A to router B", requires configuration of the policy in A and B as well as in all the routers in between. As a network evolves, the complexity of its configuration also grows, and the configuration becomes difficult to understand, extend, and debug. Patches are sometimes put into configuration files during a crisis to temporarily deal with a problem. These patches are forgotten and left in place after the pressure of the situation is lifted. Old configurations that have been replaced are not deleted, just in case the new ones are not completely debugged or to ensure that the network continues to work if the transition is incomplete. Personnel turnovers mean that configurations are edited by multiple engineers with different backgrounds and working styles. As a result of all these complications, network policies end up being configured incompletely or they contradict one another. Faulty configurations are prevalent and can lead to more than 50% of failures in computer networks and distributed systems [1]–[3]. Some of the errors can have dramatic impacts such as introducing network security vulnerabilities or leading to global connectivity disruptions. For example, misconfigurations in AS (Autonomous System) 9121 resulted in the incorrect propagation of 100K+ routes, leading to "misdirected or lost traffic for tens of thousands of networks" [4].

The complexity of a network's configuration can be compounded by ineffective configuration commands. A command is ineffective if its removal does not change the behavior of the network (e.g., a command that is never executed, or a command whose condition is always set to be false). In [5], we show that ineffective commands comprise more than 30% of the routing policy configurations in four production networks. Ineffective commands have generally been believed to be benign. Some ineffective commands are obsolete and others may be left on purpose for future use. However, we show that certain types of ineffective commands are due to operator errors leading to unexpected network behavior. Due to these errors, the behavior of the network differs from the operator's original intention. Some of these errors require prompt corrections of the configuration (e.g., filters that leak private addresses outside of the network or filters that remove intended routes).

Fig. 1 illustrates an example of ineffective BGP (Border Gateway Protocol) route filters. The line between two routers, *R1* and *R2*, represents a BGP session. The rounded rectangles, *F1* and *F2*, are route filters. The arrow in a route filter denotes the direction where the route filter is applied. *F1* is applied to *R1* for routes leaving from *R1* towards *R2*. *F2* is applied to *R2* for routes arriving at *R2* from *R1*. Each route filter has its actual commands in a rectangle connected with a dashed line. The filter has a structure similar to that of the *if-then-else* chain in a programming language. When a route arrives at the route filter, the route filter compares the route with the if-clauses sequentially. If the route matches an if-clause, the route takes the respective action, "permit" or "deny", and the

Manuscript received 19 April 2008; revised 3 November 2008. This work was supported in part by ICTI and NSF award 0756998.

S. Lee, T. Wong, and Hyong S. Kim are with the Department of Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, PA, 15213 USA (e-mail: sihyunglee@cmu.edu; tinawong@cmu.edu; kim@ece.cmu.edu).

Digital Object Identifier 10.1109/JSAC.2009.0904xx.

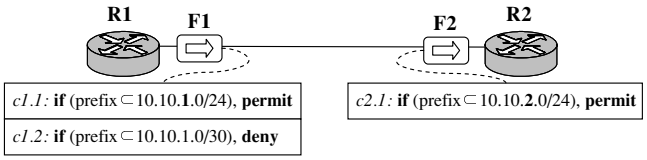


Fig. 1. A simple example configuration of inter-domain routing. A line between two routers means that there exists a BGP session between the two routers.

TABLE I
DESCRIPTION OF SYMBOLS

| Symbol | Description |
|---|---|
| *Rx* | Router |
| *Fx* | Route filter |
| *Fx(i)*, *Fx(o)* | The input and output components of *Fx*, respectively. |
| *Cx* | If-then-else chain |
| *cx*, *cx.y* | Component |
| *Sx* | Route set |
| *x:y* | BGP community |

subsequent if-clauses are not evaluated. At the end of the if-then-else chain, there is an implicit "deny" action. In Fig. 1, the first entry *c1.1* says that *F1* allows any route within the prefix 10.10.1.0/24. The next entry *c1.2* is intended to filter out the routes to the protected network 10.10.1.0/30. However, *c1.2* is ineffective: the if-clause in *c1.1* allows 10.10.1.0/30, so no route matches the condition in *c1.2*. For *c1.2* to be effective, *c1.2* needs to precede *c1.1*. A command can also be ineffective because of a command in another route filter. For example, *F1* does not allow any routes that *c2.1* matches, causing *c2.1* to become ineffective. This indicates inconsistent policy configurations at the two ends of the BGP session. In more complex cases, multiple policies contradict one another and the commands in a series of route filters cause a command to be ineffective.

Some ineffective configuration commands are actually legitimate. For example, it is common practice that the same commands are configured at multiple places, sometimes in different routers, to increase robustness in the network. To separate these legitimate ineffective commands from possible errors, we focus on two types of ineffective commands that are likely errors, out of all possible types of ineffective commands.

Our system, called NetPiler, detects ineffective commands in routing policy configurations. Routing policy configurations are one of the largest, highest impact, and the most complex parts of a network's configuration [6], [7]. NetPiler models the routing policy configuration as a program flow graph (Section III). This graph represents the set of operations that are executed in route filters and the order in which the operations are executed. It also represents the way a route propagates through the route filters in the network. Using this graph, NetPiler identifies two types of ineffective commands that are likely caused by operator errors (Section IV). The first type is *always-false predicates*, which do not match any routes. In Fig. 1, the if-clause in *c2.1* is an always-false predicate. The second type is *unreachable commands*, which are never executed. In Fig. 1, the action "deny" in *c1.2* is unreachable. The program flow graph can also be used to do "what-if scenario" testing for pending changes to configuration. We have implemented NetPiler and evaluated it on configurations from four production networks - a tier-1 nation-wide provider, two regional providers, and one university network (Section V). We show that NetPiler is powerful in detecting misconfigurations. It is able to detect a number of errors that are confirmed by the network operators. Throughout the paper, we use the symbols listed in Table I.

## II. RELATED WORK

Most of the previous work on the verification of a network configuration is based on a predefined rule set. [8] detects syntax errors within a router (e.g., undefined references) or between two end points of a protocol session (e.g., two end points of a link that participate in OSPF should be configured to have the same area.). *rcc* [9] uses a collection of network-wide policies, which are considered to be the best common practices (e.g., the internal BGP sessions should form a full mesh.). NetPiler complements these previous approaches. The errors NetPiler finds would be overlooked by these configuration checkers. The detection of ineffective components requires an accurate representation of route sets on each different route filter in the network. It also requires modification of the route sets as they propagate through the filters. FIREMAN [10] applies a flow analysis for distributed firewalls. Although their approach is similar to that of NetPiler, what constitutes an error in a firewall is different from the errors in a routing policy configuration. Also, the models for routing policies differ significantly from the models for firewalls. NetPiler models routing policies according to the routing protocol behaviors. Lastly, route filters frequently modify the attributes in a route (e.g., BGP community, AS-path, local preference, and next hop) in addition to taking two typical actions in firewalls, "permit" and "deny". This adds complexity to the analysis.

Configurations are decomposed across routers and are specified in low-level device-specific languages. These are identified as some of the major reasons that complicate the network configuration [9]. The 4D [11], RCP [12], and Ethane [13] thus propose to have a central point of decision where network-wide goals are specified by a high-level language. Although operators in these systems specify configurations in one place, there are still multiple policies to manage. Also, the operators need to learn new configuration languages and can make mistakes in high-level specifications. It may take several years before a new architecture and configuration languages become widely deployed and accepted. Our focus is to tackle the configuration problems associated with deployed legacy networks. Nonetheless, the idea of NetPiler can be applied to detect ineffective configurations in high-level specifications and to analyze the interactions among different policies. Finally, there is still ongoing research on how to map complex and various high-level objectives (e.g., Quality-of-Service and traffic engineering goals) to individual device configurations. The current high-level languages either specify a particular

aspect of a network configuration (e.g., Metarouting [14] specifies routing protocols), require the description of many low-level details (e.g., RPSL [15]), or translate simple objectives (e.g., "allow desktop group to access the server group via the proxy" in Ethane [13]).

## III. Modeling Route Filters

Our routing policy model is comprised of three different parts: models of a single route filter; models of route filters in a network of routers; and models of external and internal routing entities.

### A. Models of a Single Route Filter

A single route filter represents a routing policy that is implemented on a single router. It denotes a unit of policy as perceived by a network operator. Route filter definitions in most vendors fit the single *if-then-else* chain model (Section III-A1). Some vendors use the multiple chain model (Section III-A2).

*1) Single if-then-else Chain Model:* We begin with a brief overview of inter-domain routing and its configuration. Inter-domain routing exchanges reachability information among ASes. BGP [16] is the *de facto* standard inter-domain routing protocol. Routing policies specify the routes a router or a network accepts, filters, and forwards. A routing policy is implemented by applying route filters to BGP sessions. For example, to prevent a network from providing free transit for its settlement-free peers, one can apply a route filter on BGP sessions with each of the network's providers so that the routes from the peers are not advertised to the providers.

Although each router vendor has its own configuration language, a route filter in most major vendors (e.g., Cisco, Juniper, Avici, Quagga) can be modeled as an *if-then-else* chain. An *if-clause* describes the routes to which the policy is applied, in terms of the attributes of the routes. A *then-clause* specifies the actions on the routes such as "permit" and "deny", along with commands that manipulate the attributes of the routes. These attributes include the AS-path and the BGP community. The AS-path attribute contains a list of ASes that the route has traversed. The AS-path is used to prevent routing loops and to implement routing policies. For example, a route is not preferred if the route goes through an AS that is known to have longer delays than other ASes. A BGP community [17] refers to a group of routes with a common announcement profile. It is one of the most widely used attributes. A 32-bit value is tagged onto the routes that belong to the community.

Our model for a single route filter is an acyclic directed graph. A vertex in this graph represents a command in the filter. We call a vertex in this graph a *component*. A directed edge in this graph represents the flow of control along the components in the if-then-else chain. The model has a single entry component *input* and a single exit component *output*. The input component connects to the output component via two types of components: a *conditional component* (i.e. a predicate) or an *action component*. The input component is adjacent to the head component of the chain, the component that is evaluated first in the chain. The output component is adjacent from all the action components that are a "permit".

**Conditional component:** There is one conditional component for each prefix or route attribute (e.g., AS-path, community) in an if-clause. For example, an if-clause "if (community `100:1` exists) or (community `100:2` exists)" is decomposed into two conditional components, "if (community `100:1` exists)" and "if (community `100:2` exists)". This decomposition accurately pinpoints the location of ineffective components. For example, if there is no decomposition (i.e., if a component represents both of the two communities) and one of the two communities is ineffective, we then can detect that the component is ineffective as a whole, but we cannot identify which one of the two communities is ineffective. The conditional component has two outgoing edges, the true branch and the false branch.

**Action component:** Unlike a conditional component, an action component includes all the actions defined in the same then-clause. We do not decompose an action component as we currently do not examine the effectiveness of each individual action. An action component has no outgoing edge, if its action is a "deny". If its action is a "permit", the action component has a single outgoing edge to the output component. At the end of the chain, there is an implicit default action. This default action is either a "permit" or a "deny" and differs from vendor to vendor. It is adjacent from the false branches of the conditional components that are evaluated last in the if-then-else chain.

Fig. 2 illustrates an example of an if-then-else chain in the single route filter model. Each rectangle represents either a conditional or an action component. We append a unique id, c*, to each component. The true branch and the false branch of a conditional component are denoted by a solid line with T and a dotted line with F, respectively. Components *c3.1* and *c3.2* drop a route if its AS-path matches the regular expression 777$ (i.e., if the route originates from AS777). Components *c3.3* through *c3.5* permit a route if its destination prefix is within 10.10.0.0/16 but not within 10.10.1.0/24. (The symbol ¬ in *c3.3* negates the condition.) All the other routes are dropped by the default "deny" action, *c3.6*. Fig. 3 shows the actual configuration commands used to configure the route filter in Fig. 2.

*2) Multiple if-then-else Chain Model:* In addition to the single if-then-else chain, the Juniper and new Cisco (IOS-XR) configuration languages allow multiple if-then-else chains to be applied in tandem. Besides the actions of "permit" and "deny", JUNOS also has a few other actions to redirect the control flow in the if-then-else chains. The keyword "next term" continues to evaluate the following if-then clause. The keyword "next policy" jumps to the subsequent if-then-else chain. The language can also call (or "apply") another if-then-else chain within an if-then-else chain. When no match is found in the callee if-then-else chain, the control returns to the caller if-then-else chain. These features improve the modularity of the configuration as we can reuse common components in multiple route filters. In Fig. 4, we show an example of this multiple chain model. The filter applies two if-then-else chains in series, *C4* and *C5*. Another chain *C6* is applied in *C4*. The components in the same if-then-else chain are enclosed in the same rounded rectangle. To emphasize the functions of the three new actions, we omit the details of the

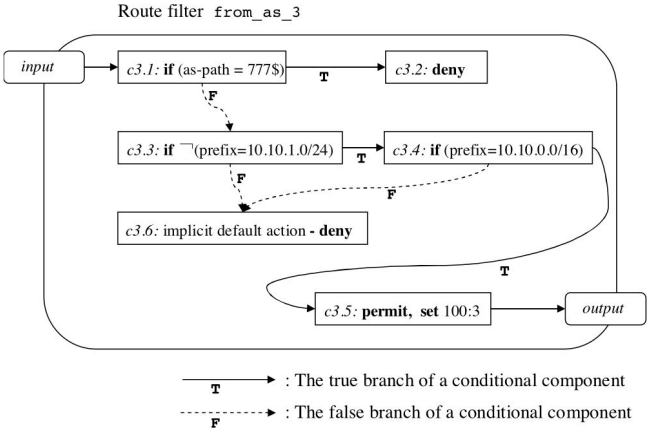


Fig. 2. An example of a route filter model.

```
01  neighbor 3.3.3.3 remote-as 3
02  neighbor 3.3.3.3 route-map from_as_3 in
03  route-map from_as_3 deny 10
04   match as-path 55
05  !
06  route-map from_as_22 permit 20
07   match ip address prefix-list 6
08   set community 100:3
09  !
10  ip as-path access-list 55 permit 777$
11  ip prefix-list 6 deny 10.10.1.0/24
12  ip prefix-list 6 permit 10.10.0.0/16
```

Fig. 3. Configuration commands used to configure the filter in Fig. 2 and to apply the filter to an external neighbor in AS3.

conditional components. We assume that the implicit default action at the end of the chains is a "deny".

### *B. Models of Route Filters in a Network of Routers*

At a high level, our model of route filters in a network represents the way routes flow through route filters in the network. In our model, a route filter *Fx* connects to another route filter *Fy* if *Fx*'s output can be re-advertised to the router where *Fy* is applied. To know how routes are re-advertised, we need to consider the BGP session-level topology as well as the rules that BGP follows to re-advertise its best route.

**BGP session-level topology:** BGP exchanges reachability information through a BGP session that runs over TCP between routers. A BGP session is called an external BGP session (eBGP) if the session is between routers in different ASes. A BGP session between routers in the same AS is called an internal BGP session (iBGP). Through iBGP, routers within the same AS share routing information that they receive over eBGP. A full mesh is created when there are iBGP sessions between every pair of routers within an AS. Although a full mesh is commonly used with small ASes, this approach does not scale as the number of router increases. One solution, called route reflector, divides the routers in an AS into a two-level hierarchy. Routers of the top-level are route reflectors and they form iBGP sessions with one another. Routers of the bottom-level are clients and each client forms an iBGP session with a single route reflector. Then, a full mesh is created only among the route reflectors.

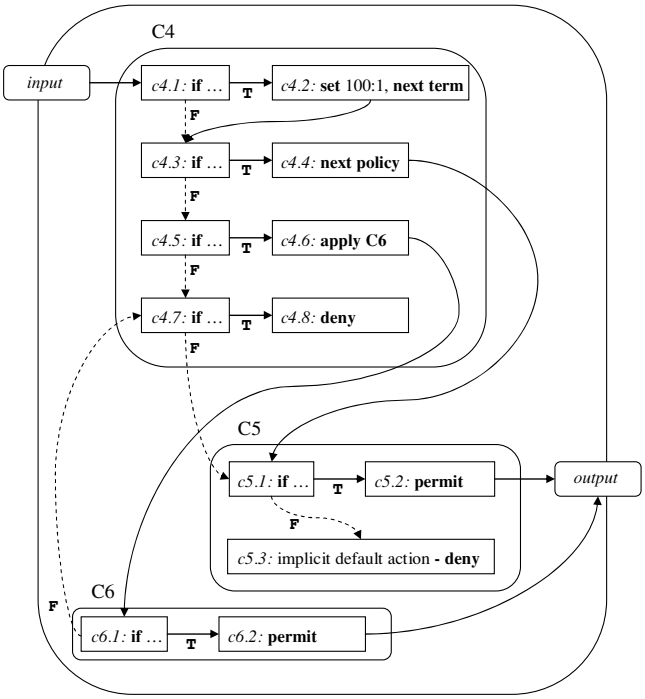


Fig. 4. An example of a route filter with multiple if-then-else chains.

**Rules to re-advertise the best route:** When a router selects the best route toward a destination, the best route is re-advertised according to the following rules. *Rule(1)*: if the best route is learned from an eBGP session or from a client iBGP session, the route is re-advertised on all other BGP sessions in the router. *Rule(2)*: if the best route is learned from a non-client iBGP session, the route is re-advertised on the client iBGP sessions and the eBGP sessions, but the route is not re-advertised to the other non-client iBGP sessions [16], [18]. In both rules, the best route is not re-advertised to the sender of the route to prevent routing loops.

*1) Creating a Model of Route Filters in a Network of Routers:* Our model of route filters in a network has an inbound filter and an outbound filter on every BGP session configured on a router. These filters represent the import policy and the export policy applied to the BGP session, respectively. If no filter is configured, the corresponding component in the model has a single action, a "permit". This default behavior allows every route. Route filters are linked to one another according to the BGP session-level topology and re-advertisement patterns as described above. Let *F(i)* and *F(o)* denote the input and output components of route filter *F*, respectively.

**Connection of filters in the same router:** Let *Fx* denote an inbound filter in router *R*. *Fx* is on a BGP session with neighbor *Rx*. Let *Fy* denote an outbound filter in the same router *R*. *Fy* is on a BGP session with neighbor *Ry*. The model has a directed edge (*Fx(o)*, *Fy(i)*) if the best route learned from *Rx* is re-advertised to *Ry* according to rules (1) and (2) in Section III-B.

**Connection of filters in different routers in the same network:** Let router *Rx* have an iBGP session with router *Ry*. *Rx* has an outbound filter *Fx* on the session. *Ry* has an inbound filter *Fy* on the session. Then, the model has a directed edge

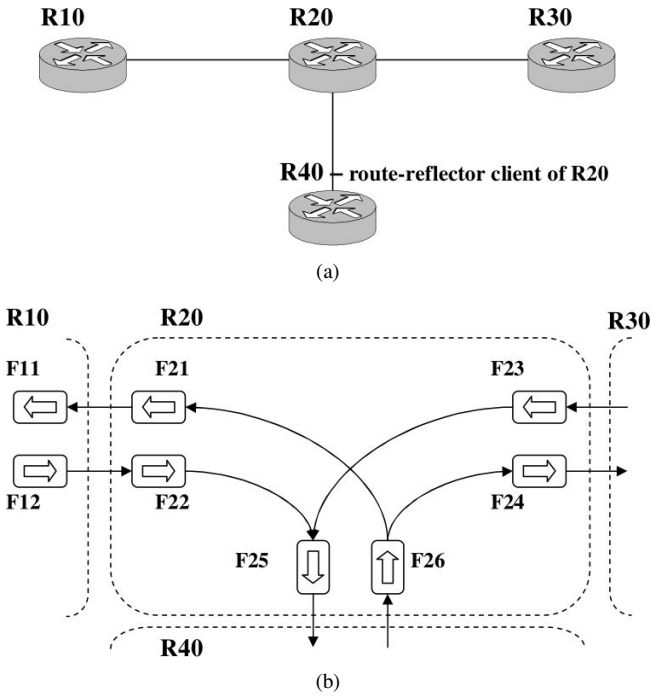


Fig. 5. An example network (a) and its corresponding model for router *R20* (b). Note that the model in (b) does not include the external and internal routing entities.

(*Fx(o)*, *Fy(i)*).

Fig. 5(a) illustrates an example network. *R20* is the route reflector of its client *R40*. *R20* has non-client iBGP sessions with *R10* and *R30*. Fig. 5(b) shows the corresponding model for router *R20*. For simplicity, we do not show the components within individual route filters. According to rule (1), *F26* is adjacent to all other outbound filters in *R20*. According to rule (2), *F22* and *F23* are not adjacent to each other, but they are adjacent to *F25*. Note that in order to complete this model, we need to add the model of external and internal routing entities, as described in the next section.

### C. Models of External and Internal Routing Entities

Routes are injected into the BGP process of a router externally as well as internally. The BGP process can also re-advertise its routes to external neighbors or other internal routing processes. In this section, we include these external and internal routing entities in our model.

**Models of an external neighbor:** Let router *R* in our AS have a BGP session with an external neighbor. Similar to an iBGP session, the model has an inbound filter *Fx* and an outbound filter *Fy* on this eBGP session. According to Rule (1), *Fx* is adjacent to all the outbound filters in the same router, except *Fy*. *Fy* is adjacent from all the inbound filters in the same router, except *Fx*. The external neighbor is modeled as two components: 1) a component that represents an entity that injects its routes into *R* and is adjacent to *Fx(i)*; 2) a component that represents an entity that receives routes advertised from *R* and is adjacent from *Fy(o)*. The two separate components indicate that the external neighbor will not re-advertise a route back to the sender of the route.

**Models of an internal routing entity:** We model other routing processes and static routes in the same manner as we model an external neighbor (i.e. as if the router has an eBGP session with each of these internal entities.). The model has an inbound filter and an outbound filter on each of these sessions. Each entity is represented by two separate components. The inbound and outbound filters limit the redistribution of the routes both from and to these routing entities. If no redistribution is configured, the filter consists of a single action, a “deny”. The inbound filter typically adds BGP communities to the routes. These communities indicate the origin of the routes. NetPiler detects the ineffectiveness of these filters. These filters connect with the other filters in the same manner as a filter on an eBGP session is connected. The only exception is that the filters on an internal routing entity do not connect with the filters on another internal routing entity. We do not model the interactions among these internal routing entities (e.g., the route redistribution between the OSPF process and the RIP process).

Fig. 6 illustrates an example of external and internal routing entities. There are two internal routers, *R60* and *R70*. Both *R60* and *R70* have an eBGP neighbor, *R50* and *R80*, respectively. Static routes are configured in *R60*. The outbound filter to the static routes, *F65*, is a placeholder and is not used. Static routes are only injected into a routing process but do not receive any routes. Note that this completed model is an acyclic directed graph. A loop in the model means a potential routing loop. NetPiler reports any loops found in the model.

## IV. Detection of Ineffective Components

Given the model of route filters, we annotate each edge in the model with the set of all the routes that can appear on the edge. On an edge between two route filters, this set represents all the routes that can be advertised from one filter to the other. On an edge between two components within a route filter, this set represents all the routes that can be evaluated against the conditions in the components or the routes that can take actions as specified in the components. Although only a subset of these routes can appear on the edge at one time, we consider the entire set of routes in order to identify the components that are ineffective in any possible state of the network. This process of annotating the edges is defined as *range propagation* since we annotate the edges with the range of routes as if the routes propagate through the network. During the range propagation, we identify two types of ineffective components: unreachable components and always-false predicates. An unreachable component has an empty input route set (i.e. its incoming edge is annotated with an empty set). The control never reaches the component. An always-false predicate is a conditional component that has an empty output route set to its true branch - the predicate does not match any input. We describe these methods of detection in Section IV-A. In Section IV-B, we explain how we represent the route set. The correct representation of the route set determines the accuracy of the detection.

### A. Methods of Detection

A route entering a network either originates from a router in the network or is received from an external network. The route is then re-advertised to other routers in the same network

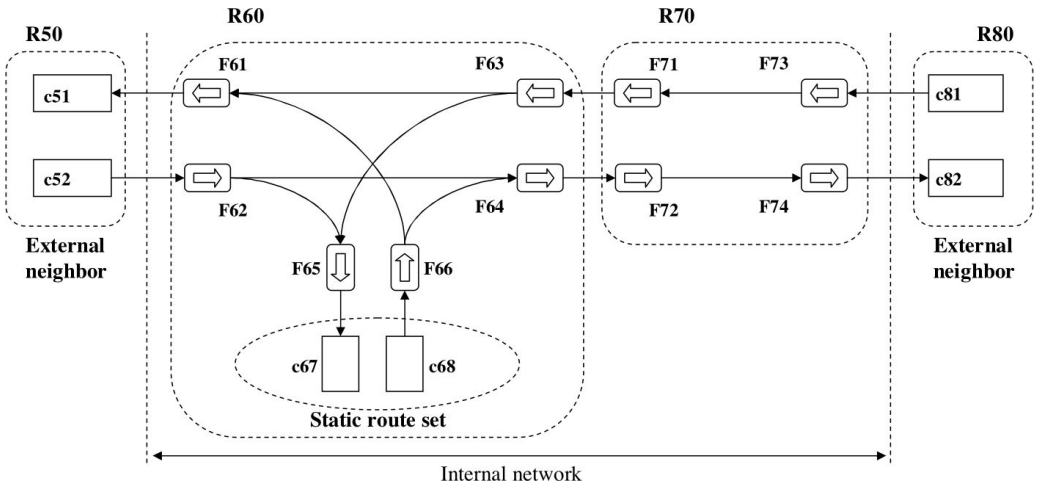


Fig. 6. An example of a model of route filters in a network. Routers *R60* and *R70* are internal routers. Each of *R60* and *R70* has an eBGP session with the external neighbors *R50* and *R80*, respectively. Static routes are configured in *R60*.

or to other external networks. The range propagation describes this journey of the route. It begins by the injection of external routes as well as internal routes into the network (Section IV-A1). As we propagate these routes in each route filter across the network (Section IV-A2), the routes propagate through each component in the filter (Section IV-A3). While we propagate the routes, we update the edges with the range of the routes. We also detect ineffective components at the same time (Section IV-A4).

*1) Injection of Routes from External and Internal Routing Entities:* Each external and internal routing entity *cx* has a set of routes *Sx* which *cx* injects into the network. We annotate each outgoing edge of *cx* with *Sx*. We determine *Sx* as follows. i) If *cx* is a static routing entity, *Sx* is clearly defined in the configuration. ii) If *cx* represents an external neighbor or an internal routing process, unless we know the exact properties of *Sx* (or unless we want to test a particular scenario where a certain set of routes is injected), we do not make any assumption about these properties except about those we know for certain, such as the last AS that the route has traversed (e.g., the last AS in the AS-path in the route from the external AS3 is 3.). We may miss some of the misconfigurations because we do not know the exact properties in advance, but we do not falsely classify a component as a misconfiguration (i.e. there is no false positive.).

*2) Range Propagation across Route Filters in a Network of Routers:* The algorithm for the range propagation is presented in Fig. 7. The range propagation is done one filter at a time. A filter *Fx* is *ready for the range propagation* if every incoming edge of *Fx* is annotated with the range of routes. A queue *Q* holds the filters that are ready for the range propagation. Before the algorithm begins, *Q* initially has all the filters that are adjacent from an external or internal routing entity. These filters are ready for the range propagation since their incoming edges are annotated as the result of the process in Section IV-A1. The while-loop pops each filter *Fx* from *Q* and propagates the range of routes within *Fx*. We describe how this is done in Section IV-A3. The input $Fx_{\cdot input}$ to *Fx* (i.e. the range of routes that propagate into $Fx$) is the union of all the ranges of routes annotated on the incoming edges of $Fx$. As a result of the range propagation within $Fx$, $Fx$ has the output $Fx_{\cdot output}$, which is the subset of the routes in $Fx_{\cdot input}$ that $Fx$ permits. We then annotate every outgoing edge $e$ of $Fx$ with $Fx_{\cdot output}$. During this annotation, if $e = (Fx, Fy)$ (i.e., if $e$ is incident with a filter $Fy$ ($y \neq x$) and not incident with an external/internal routing entity) and if $Fy$ is ready for the range propagation, we insert $Fy$ into $Q$. The loop finishes when $Q$ is empty. Since the model is acyclic and connected, every filter propagates the range of routes once. Thus, the running time of the algorithm is linearly proportional to the number of filters in the model.

$M$, $E(M)$: the network model and its edge set.
$Q$: the queue of the filters to propagate the range. $Q$ initially has all the filters adjacent from an external or internal routing entity.
$Fx$: the $x^{th}$ filter in $M$.
$Fx_{\cdot num\_incoming\_edges}$: the indegree of $Fx$.
$Fx_{\cdot num\_annotated\_incoming\_edges}$: # incoming edges annotated with the range of routes. It is initialized to 0.
$Fx(i)$, $Fx(o)$: input and output components of $Fx$, respectively.
$Fx_{\cdot input}$: the input to $Fx$ (i.e. the range of routes that propagate into $Fx$).
$Fx_{\cdot ouput}$: the output of $Fx$ (i.e. a subset of routes in $Fx_{\cdot input}$ that $Fx$ permits).

```
00  range_propagation_across_the_network () {
01    while (|Q| > 0) {
02      Fx = Q.pop()
03      Fx.range_propagation_within_the_filter()
04      for each Fy s.t. (Fx(o), Fy(i)) ∈ E(M)
05        Fy.input=Fy.input ∪ Fx.output
06        if (Fy.num_incoming_edges=++Fy.num_annotated_incoming_edges)
07          Q=Q ∪ Fy
08    }
09  }
```

Fig. 7. Algorithm for range propagation across a network.

Table II presents each iteration of the while-loop for the model in Fig. 6. The bold-faced filters represent the ones that are ready for the range propagation and are inserted into $Q$ in the iteration. Before the while-loop starts (i.e. #iteration=0), the outgoing edges from the external and internal routing entities, $c52$, $c68$, $c81$, are annotated, and their adjacent filters $F62$, $F66$, $F73$ are inserted into $Q$. At each iteration, the loop pops a filter $Fx$ from $Q$, propagates the range within

TABLE II
ILLUSTRATION OF THE APPLICATION OF THE ALGORITHM IN FIG. 7 ON THE MODEL IN FIG. 6

| # iterations in while() loop | Filter *Fi* that propagates the range | Annotated edges in the iteration | $Q$ |
|---|---|---|---|
| 0 | | (*c52*, ***F62***)<br>(*c68*, ***F66***)<br>(*c81*, ***F73***) | {***F62***, ***F66***, ***F73***} |
| 1 | *F62* | (*F62*, *F65*)<br>(*F62*, *F64*) | {*F66*, *F73*} |
| 2 | *F66* | (*F66*, *F61*)<br>(*F66*, ***F64***) | {*F73*, ***F64***} |
| 3 | *F73* | (*F73*, ***F71***) | {*F64*, ***F71***} |
| 4 | *F64* | (*F64*, ***F72***) | {*F71*, ***F72***} |
| 5 | *F71* | (*F71*, ***F63***) | {*F72*, ***F63***} |
| 6 | *F72* | (*F72*, ***F74***) | {*F63*, ***F74***} |
| 7 | *F63* | (*F63*, ***F61***)<br>(*F63*, ***F65***) | {*F74*, ***F61***, ***F65***} |
| 8 | *F74* | (*F74*, *c82*) | {*F61*, *F65*} |
| 9 | *F61* | (*F61*, *c51*) | {*F65*} |
| 10 (# filters in the network) | *F65* | (*F65*, *c67*) | {} |

$Fx$, and annotates the outgoing edges of $Fx$ with the output of the range propagation. If a filter $Fy$ adjacent from $Fx$ is ready for the range propagation, $Fy$ is inserted into $Q$. The loop ends when every filter propagates the range.

*3) Range Propagation in a Single Route Filter:* Each individual filter model is loop-free regardless of whether it has a single if-then-else chain or multiple if-then-else chains. Thus, we can use the same algorithm to propagate the range through the components within the filter as we propagate the range through the filters across a network. When the algorithm calls *range_propagation_within_the_filter()* (line 03, Fig. 7), we begin to propagate the range in the head component of the filter. The incoming edge to the head component is annotated with the input to the filter. Each of the other components is inserted into queue $q$ whenever all of its incoming edges are annotated with the range. $q$ contains all the *components* that are ready for the range propagation. We pop each component $c_i$ from $q$ and propagate the range within $c_i$. This propagation annotates the outgoing edges of $c_i$. We then examine the effectiveness of $c_i$ according to the definitions in Section IV-A4. The propagation in the filter ends when $q$ is empty. As a filter propagates the range only once, so does each component in the filter. We propagate the range within the component $c_i$ as follows.

**Range propagation in a conditional component:** A predicate represents a collection of routes *Sc* that matches the predicate. Let *St* and *Sf* refer to the ranges of routes annotated on the true branch and the false branch of *cx*, respectively. Let *Si* denote the union of all the ranges annotated on the incoming edges of *cx*. Then, $St = Sc \cap Si$, and $Sf = Si - Sc$.

**Range propagation in an action component:** If the action is a "deny", the propagation ends. If the action is a "permit", *Si* is modified according to the additional actions configured to manipulate the attributes of the routes. The result of the modification is annotated on the single outgoing edge of the component. The output of the filter is the union of all these ranges annotated on the outgoing edges of the action components.

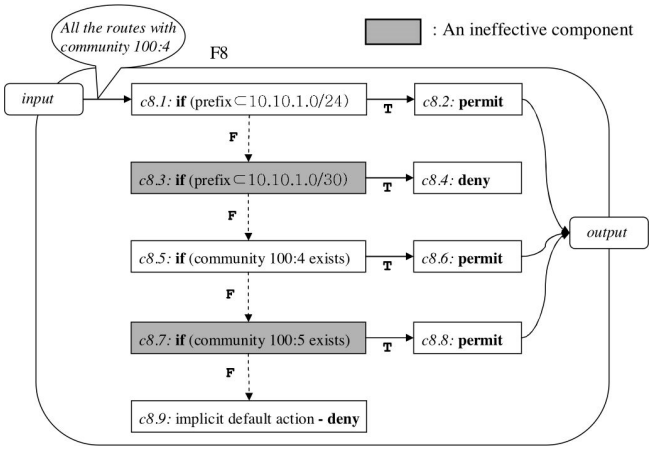


Fig. 8. An example of a route filter with two ineffective components. The input to the route filter is a set of all the routes tagged with community `100:4`.

*4) Detection of Ineffective Components:* A conditional component *cx* is an *always-false predicate* if its *St*={}. An action (or conditional) component *cy* is *unreachable* if its *Si* ={} (i.e. *cy* is unreachable if all the ranges in the incoming edges are empty.). We present an example of these two types of ineffective components in Fig. 8. The input to the filter *F8* is a set of routes with community `100:4`. *c8.3* is an always-false predicate since *c8.1* matches a superset of the routes that *c8.3* matches. The rest of the routes are matched by *c8.5*, causing *c8.7* to be unreachable. Note that *c8.7* is unreachable according to the input to *F8*, whereas *c8.3* is ineffective regardless of the input to *F8*. As the input depends on the configuration in the other elements of the network, so does the ineffectiveness of *c8.7*. The ineffectiveness of *c8.3* is determined only by the configuration in *F8*. The strength of NetPiler is that it examines the flow of routes throughout a network, thereby allowing it to detect ineffective components caused by other elements in the network as well as ineffective components caused by the components within the same filter.

We do not flag one class of always-false predicates, called generalization, that are most likely to be valid configurations [10]. In the context of routing policy, generalization is typically used to act on a set of routes except its small subset. For example, "if (prefix ⊂ `prefix-list1`), deny" followed by "if (prefix ⊂ `prefix-list2`), permit" is a generalization, if `prefix-list1` matches 10.10.1.0/24, and `prefix-list2` matches 10.10.1.0/24 or 10.10.2.0/24. The detection of a generalization is straightforward, and the details can be found in the technical report version of this paper [19]. The technical report also lists the types of ineffective components that are legitimate and thus not considered by NetPiler.

### *B. Representation of Route Sets*

The range propagation and detection of ineffective components require an accurate representation of route sets. A conditional component performs set operations on these route

sets. An action component modifies these sets. We also evaluate whether these sets are empty in order to detect the ineffectiveness of the component. We use a Binary Decision Diagram (BDD) [20] to represent the route sets. A BDD provides a compact representation of the set, and most of the BDD libraries fully support the set operations and the modifications of the set. Details on this representation can be found in our technical report [19].

## V. Evaluation

We have implemented NePiler and evaluated it on the configurations from four production networks. The characteristics of the four networks are summarized in Table III. Due to the proprietary information of these networks, we only characterize the networks into small, medium, and large: a small network has less than 10 routers; a medium network has between 11 and 50 routers; and a large network has over 50 routers. In these networks, routing policy configurations are a major portion, comprising up to 70% of the configurations. The NetPiler implementation is written in C++ and has approximately 128,000 lines of code. The total running time is less than one minute in the university network, and less than 5 minutes in a large provider network with thousands of filters. The NetPiler code can be further optimized if faster run time is desired. The algorithm is linearly proportional to the number of components. The running time also depends on the complexity of the commands that manipulates the route set (e.g., the addition and deletion of communities, and the modification of AS-path). For example, provider 2 adds and deletes from 15 to 20 communities in an action component, whereas provider 3 adds mostly one community.

For three networks, we analyze the daily snapshots over a two-year period to study the order in which the ineffective components are configured. This helps classifying the results into absolute errors, possible errors, or benign components. NetPiler detects ineffective components in the configurations of 101 BGP sessions. Table IV summarizes the results. More than half of these sessions are likely to contain errors and 28 sessions are confirmed to have operator errors. Some of these ineffective components represent critical errors that allow private addresses to be advertised externally and errors that filter out intended routes.

### A. Error Classification and Count

The detected errors are classified into absolute errors, possible errors, or benign components, denoted by *AE*, *PE*, and *B* in Table IV, respectively. Absolute errors are divided into two categories: i) ineffective components confirmed by the operators as errors, and ii) ineffective components that we are certain of their misconfigurations. Possible errors are ineffective components that are not verified by the operators but are likely to be errors. The operators are usually limited to their knowledge of the configurations at that time, and they are not necessarily aware of entire history of configuration changes. For this type of error, we provide more explanations about errors. Benign components are ineffective components that are detected by NetPiler but which do not have any negative impact on the network. However, some of these benign components can complicate the configuration and confuse the operator if not corrected in a timely manner.

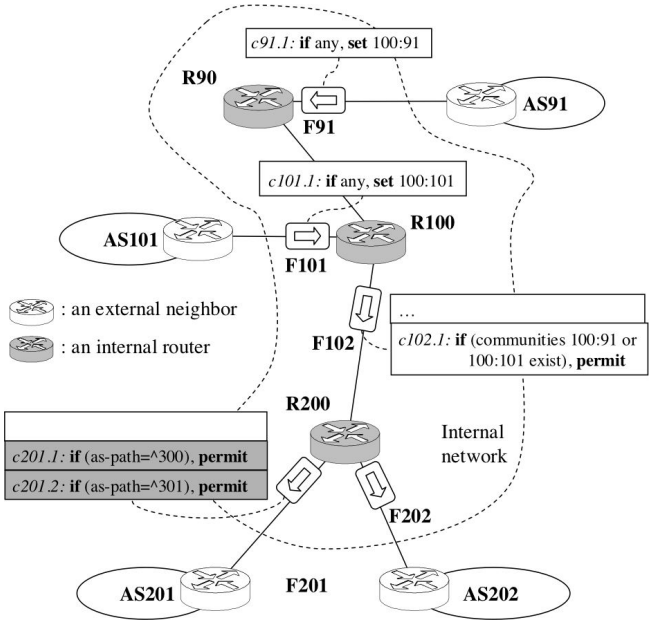


Fig. 9. Ineffective components caused by a tandem of remote filters.

We count the number of BGP sessions that contain each type of error, as the number of sessions represents the impact of the errors. We do not count each individual ineffective component since multiple ineffective components can signal the same error (e.g., one unreachable component causes all the subsequent components in the same if-then-else chain to be unreachable). Also, we do not report the numbers for each network that we tested to protect the proprietary information of these networks.

### B. Results

In this section, we describe each type of ineffective components that NetPiler detected, in the order they appear in Table IV.

*1) Absolute and Possible Errors Caused by Multiple Filters in a Network:*

**Ineffective components caused by a series of remote route filters:** Fig. 9 illustrates a case of a series of route filters causing a route filter to be ineffective. For simplicity, we present an if-then clause in one single component. A route reflector, *R100*, advertises a certain set of routes $S_{201}$ to its client, *R200*, via *c102.1*. This set is with communities `100:91` or `100:101`. These communities are attached to the routes from AS91 and AS101, respectively. The routes in $S_{201}$ are supposed to be re-advertised to AS201 via *c201.1* and *c201.2*. However, *c201.1* and *c201.2* do not match any routes in $S_{201}$ and become ineffective since these two components match routes only from AS300 and AS301. The operator confirmed that *c201.1* and *c201.2* are misconfigurations. The misconfigurations are not immediately visible since there is another session to AS201 at a different router. However, if this other session fails, the destinations corresponding to $S_{201}$ will become unreachable from AS201.

TABLE III
SUMMARY OF DATASET AND PERFORMANCE CHARACTERISTICS

| Network | Size | # BGP sessions per router[b] | # filters, # components | Elapsed Time[a] | |
|---|---|---|---|---|---|
| | | | | Model Construction (written in C++) | Range Propagation (written in C++) |
| Univ. 1 | Large | (3,3,7) | (449,1286) | Less than 00:01 | Less than 00:01 |
| Provider 1 | Large | (3,45,66) | (3982,18407) | 00:18 | 03:29 |
| Provider 2 | Small | (11,22,25) | (228,3378) | 00:02 | 01:49 |
| Provider 3 | Medium | (3,17,202) | (1558,11688) | 00:12 | 00:33 |

[a]Elapsed Time in minutes:seconds. The times were observed when running the components on a machine with a CPU of 2.8 GHz and 2Gb of memory.
[b]The three numbers under this column denote the $10^{th}$-percentile, median, and the $90^{th}$-percentile values, respectively.

TABLE IV
SUMMARY OF NETPILER RESULTS

| Range of the problem | Problem | AE[a] | PE | B | Possible Negative Impacts |
|---|---|---|---|---|---|
| Ineffective components caused by multiple filters in a network | Ineffective components caused by multiple remote filters | 1 | | | A peer loses connection to a set of destinations upon a session failure. |
| | Incorrect prefix lengths on static routes | 3 | | | Prevent static routes from being advertised to the network. |
| | Inconsistent policies on two ends of a session | | 8 | | Drop intended routes. Overload routers. |
| | Impossible Boolean conditions | | 6 | | Drop intended routes. |
| Ineffective components caused by a single filter | Double Negations | 17 | | | Leak internal routes. Accept bogus routes. |
| | Shadowing different actions | 3 | 3 | | Drop intended routes. |
| | Shadowing the same action | | 40 | | Provide free transit to a peer. |
| | Typos in as-path definitions | | 1 | | Drop intended routes. |
| | Missing definitions | 4 | | | |
| | Repetition of identical components | | | 13 | Can complicate the configurations. |
| | Reuse of common components | | | | |
| Total | | 28 | 58 | 15 | |

[a]AE, PE, and B denote 'Absolute Error', 'Possible Error', and 'Benign Components', respectively.

**Minsconfigured prefix length:** In one network, a set of static routes *Sx* is defined and re-advertised to other networks via filter *Fx*. In the definition of the static routes, the prefix lengths of a few routes in set *Sx* are not configured. These routes are thus assigned a default prefix length of 32, which is different from their correct prefix lengths. The prefix lengths are correctly configured in *Fx*. As a result, *Sx* do not match *Fx*, causing the respective predicates in *Fx* to be ineffective. This means that the static routes will not be re-advertised properly.

**Inconsistent policies in tandem:** Fig. 1 illustrates this case. Two filters *F1* and *F2* applied at the two ends of a BGP session permit different sets of routes, *S1* and *S2*, respectively. We observe two different situations: i) $S1 \subset S2$ and ii) *S1* and *S2* are not a subset of each other. In some cases of i), $S1=\{\}$ (i.e. *F1* drops every route). NetPiler reports the always-false predicates in *F2* that match the routes in $S2-S1$. The operators are concerned about these inconsistent policies. Some operators want to remove *F2* in the case of i). *R2* receives the same set of routes regardless of the application of *F2*. This removal of *F2* clarifies the configuration. It also eliminates the need to evaluate the ineffective filter *F2*, which adds to the computation load in *R2*.

**Impossible Boolean conditions:** In the inbound filters on several eBGP sessions, an if-then clause matches routes with the AS-path of length 0. This condition can never evaluate to true. When an AS advertises a route externally, the AS prepends its AS number in the AS-path of the route, and the length of the AS-path becomes greater than 0. These components may not be misconfigured but are simply deactivated.

*2) Absolute and Possible Errors Caused by a Single Filter:*

**Double Negation:** A bogon prefix-list is a list of IP addresses that are private or that are not allocated. This list is used to prevent those addresses from being advertised into the network and also to the outside of the network. In one network, a bogon-list `bogon` was used with an if-clause "if (prefix $\subset$

bogon), deny". However, a wrong command "deny" is used in the definition of bogon, "ip prefix-list bogon deny 10.0.0.0/8", negating the condition. Consequently, the if-clause leaks the private routes into external networks. NetPiler detects the predicate as ineffective. This mistake comes from the fact that two different commands are syntactically the same. The keyword "deny" is used both in a command to negate a predicate and the one that drops routes. The double negation means that the routes are not denied. The same mistake is found in another route filter that is supposed to drop all the routes. Because of the mistake, the ineffective filter allows every route.

**Shadowing:** *Shadowing* occurs when an if-then clause matches a superset of the entire set of routes that the subsequent if-then clause matches, shadowing the latter clause [10]. The latter clause becomes ineffective. The two clauses either have the same action or different actions. In Fig. 8, *c8.1* shadows *c8.3*. *c8.3* should precede *c8.1*. We also find a shadowing within an AS-path list (or a prefix list). An AS-path shadows a subsequent AS-path as shown in the following commands.

```
1 ip as-path access-list 77 permit _200_
2 ip as-path access-list 77 permit _232_
3 ip as-path access-list 77 permit _300_
4 ip as-path access-list 77 permit ^300$
```

The AS-path list is evaluated sequentially. The comparison returns when the first match is found. The regular expression '_300_' (i.e. 300 appears anywhere in the AS-path) in line 3 shadows '^300$' (i.e. 300 is the only AS in the AS-path, indicating that the route is originated from 300.) in line 4. When studying the daily snapshots of the configurations, we find that only line 1 and 3 appear in the initial snapshot. We also observe that line 2 and 4 are added together in a later snapshot while only line 2 is effective. If line 4 is intended to override line 3, the impact of this possible misconfiguration is to advertise a larger set of routes '_300_' to a peer and provides a free transit to a number of destinations.

**Typos and missing definitions:** Another type of error happens when there is a typo in an AS-path. In one case, an if-then-else chain on an eBGP session permits a route to the subsequent chains if the route matches an AS-path '^400$', which is unlike the convention in all the other eBGP sessions in the same peer group, '^400' (i.e., 400 is the first AS in the AS-path, meaning that the route is re-advertised from AS400.). This allows only a small collection of routes that originate from AS400. As a result, many of the subsequent if-then-else chains become ineffective. The other type of possible errors is missing definitions, which occurs when an if-then clause matches a prefix list that does not exist. Instead, a community list of the same name is defined. Since the definition is missing, the if-then clause matches every route, causing the subsequent if-then clauses to be unreachable. Other configuration checkers can more efficiently detect this type of error [8], [9].

*3) Benign Components:*

**Identical predicates:** Two equivalent community lists 99 and clist2 are used in an if-clause, "if (any community in 99 exists) or (any community in clist2 exists)". The second predicate is ineffective. The network is in the process of replacing the numbered community list 99 with the named community list clist2. The former community list should be removed in a timely manner as soon as the update is complete. Otherwise, these obsolete configurations can impair the readability of the configuration. We also observe cases where one if-then clause is followed by a meaningless if-then clause (e.g., two consecutive if-then clauses that deny every route, or an if-then clause that denies every route followed by one that permits no route.). These can also confuse the operators and possibly lead to further configuration errors. Therefore, we suggest removing the latter if-then clause.

**Reuse of common components:** The operators configure a few if-then-else chains in tandem, knowing that a part of one chain *Cx* can be made ineffective by a preceding chain. *Cx* is simply reused rather than defining another chain that contains no ineffective components.

## VI. Conclusion

We propose NetPiler, a system that detects misconfigurations in a routing policy configuration. As opposed to other syntactic configuration checkers, NetPiler accurately models the behavior of the configuration in a similar fashion to that of a compiler in a programming language. The model represents the way that routes flow and are processed across a network. This model significantly expands the range of misconfigurations that we can detect, from errors that require simple syntactic analysis to errors that require complete semantic evaluation. NetPiler evaluates the routing policies in four production networks and it successfully detects a number of serious misconfigurations quickly. One of the critical errors is found in the route filters that block the announcement of intended routes or ones that announce private addresses externally. Many of these configuration errors can be overlooked by other configuration checkers. We believe that NetPiler can be used for semantic verification of the configuration with many possible requirements in addition to detecting ineffective commands that we study in this paper.

## Acknowledgment

We thank Kyriaki Levanti for all her comments and suggestions. We would also like to express our gratitude to the anonymous reviewers and the guest editors for all their comments.

## References

[1] R. Mahajan, D. Wetherall, and T. Anderson, "Understanding BGP misconfigurations", in *Proc. SIGCOMM*, Aug. 2002.
[2] D. Oppenheimer, A. Ganapathi, and D. Patterson, "Why do Internet services fail, and what can be done about it?" in *Proc. USITS*, 2003.
[3] A. Wool, "A quantitative study of firewall configuration errors",*IEEE Computer*, June 2004.
[4] A. C. Popescu, B. J. Premore, and T. Underwood, "Anatomy of a leak: AS9121", NANOG 34, May 2005.
[5] S. Lee, T. Wong, and H. S. Kim, "Improving dependability of network configuration through policy classification", in *Proc. IEEE DSN*, Jun. 2008.
[6] D. Maltz, G. Xie, J. Zhan, H, Zhang, G. Hjalmtysson, and A. Greenberg, "Routing design in operational networks: a look from the inside", in *Proc. ACM SIGCOMM*, Aug. 2004.

[7] S. Lee, T. Wong, and H. S. Kim, "To automate or not to aut
on the complexity of network configuration", in *Proc. IEEE ICC*
2008.
[8] A. Feldmann and J. Rexford, "IP network configuration for intrac
traffic engineering", *IEEE Network Mag.*, 2001.
[9] N. Feamster and H. Balakrishnan, "Detecting BGP configuratior
with static analysis", in *Proc. NSDI*, May 2005.
[10] L. Yuan, J. Mai, Z. Su, H. Chen, C. Chuah, and P. Moh
"FIREMAN: A toolkit for firewall modeling and analysis", ir
*IEEE Symposium on Security and Privacy*, 2006.
[11] A. Greenberg, G. Hjalmtysson, D. Maltz, A. Myers, J. Rexford, G
H. Yan, J. Zhan, and H. Zhang, "A clean slate 4D approach to n
control and management", *ACM SIGCOMM Computer Communi
Review*, vol. 35, no. 5, Oct. 2005.
[12] M. Caesar, D. Caldwell, N. Feamster, J. Rexford, A. Shaikh,
van der Merwe, "Design and implementation of a Routing C
Platform", in *Proc. NSDI*, May 2005.
[13] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown, and S.
Shenker, "Ethane: Taking control of the enterprise", in *Proc. ACM
SIGCOMM*, Aug. 2007.
[14] T. Griffin, and J. L. Sobrinho, "Metarouting" in *Proc. ACM SIGCOMM*,
Aug. 2005.
[15] C. Alaettinoglu, C. Villamizar, E. Gerich, D. Kessens, D. Meyer, T.
Bates, D. Karrenberg, and M. Terpstra, Routing Policy Specification
Language (RPSL), RFC-2622, 1999.
[16] Y. Rekhter, T. Li, and S. Hares, A Border Gateway Protocol 4 (BGP-4),
RFC-4271, Jan. 2006.
[17] R. Chandra, P. Traina, and T. Li, BGP communities attribute, RFC-1997,
Aug. 1996.
[18] T. Bates, R. Chandra, and E. Chen, BGP route reflection - an alternative
to full mesh IBGP, RFC-2796, Apr. 2000.
[19] S. Lee, T. Wong, and H. S. Kim, "Network Router Configuration
Management", CMU Technical Report, CMU-CyLab-08-013, 2008.
[20] R. E. Bryant, "Graph-based algorithms for Boolean function manipula-
tion", *IEEE Trans. Comput.*, vol. 35, no.8, 1986.

**Sihyung Lee** (S'08) received the B.S. and M.S. degrees in electrical engineering from the Korea Advanced Institute of Science and Technology (KAIST), Daejon, Korea in 2000 and 2004, respectively, and is working toward the Ph.D. degree at the Electrical and Computer Engineering Department, Carnegie Mellon University, Pittsburgh, PA.

He worked in the Network Management group at Cisco, San Jose, CA, in the summer of 2006. He is currently working on simplifying network management. His research interests include Internet routing, routing security, network management and measurement. His work is supported by Samsung Scholarship Foundation.

**Tina Wong** received the B.S. degree (1995) with distinction in computer science from the University of Washington, Seattle, in 1995, and the M.S. and Ph.D. degrees in computer science from the University of California at Berkeley in 1998 and 2000, respectively.

She is a faculty member in the Information Networking Institute and a System Scientist at CyLab and ECE, all part of Carnegie Mellon University, Pittsburgh, PA. From 2000 to 2002, she was a Research Scientist at Hewlett Packard Laboratories, and worked in collaboration with NTT DoCoMo on Streaming media services for next-generation mobile devices. From 2002 to 2004, she was a member of the Network Science Department at Packet Design, Palo Alto, CA, during which she worked with a stellar group of people on various aspects of routing analytics, including a product. Her current research interests include simplifying network management, protecting the Internet routing infrastructure, and secure management of sensor networks.

**Hyong S. Kim** has been with Carnegie Mellon University (CMU), Pittsburgh, PA, since 1990, where he is currently the Drew D. Perkins Chaired Professor of Electrical and Computer Engineering. His primary research areas are advanced switching architectures, fault-tolerant, reliable, and secure network architectures, and optical networks. His Tera ATM switch architecture, developed at CMU, has been licensed for commercialization. In 1995, he founded Scalable Networks, a Gigabit-Ethernet switching startup. Scalable Networks was acquired by FORE systems in 1996. In 2000, he founded AcceLight Networks, an optical switching startup, and was CEO until 2002. He is an author of over 80 published papers and holds more than 10 patents in networking technologies.

Dr. Kim was an editor for the IEEE/ACM TRANSACTIONS ON NETWORKING from 1995 to 2000. He was the recipient of the National Science Foundation Young Investigator Award in 1995.