Campion: Debugging Router Configuration Differences

Alan Tang
UCLA
atang42@cs.ucla.edu

Siva Kesava Reddy Kakarla
UCLA
sivakesava@cs.ucla.edu

Ryan Beckett
Microsoft
ryan.beckett@microsoft.com

Ennan Zhai
Alibaba
ennan.zhai@alibaba-inc.com

Matt Brown
Intentionet
matt@intentionet.com

Todd Millstein
UCLA / Intentionet
todd@cs.ucla.edu

Yuval Tamir
UCLA
tamir@cs.ucla.edu

George Varghese
UCLA
varghese@cs.ucla.edu

Abstract

We present a new approach for debugging two router configurations that are intended to be behaviorally equivalent. Existing router verification techniques cannot identify all differences or localize those differences to relevant configuration lines. Our approach addresses these limitations through a *modular* analysis, which separately analyzes pairs of corresponding configuration components. It handles all router components that affect routing and forwarding, including configuration for BGP, OSPF, static routes, route maps and ACLs. Further, for many configuration components our modular approach enables simple *structural equivalence* checks to be used without additional loss of precision versus modular semantic checks, aiding both efficiency and error localization. We implemented this approach in the tool Campion and applied it to debugging pairs of backup routers from different manufacturers and validating replacement of critical routers. Campion analyzed 30 proposed router replacements in a production cloud network and proactively detected four configuration bugs, including a route reflector bug that could have caused a severe outage. Campion also found multiple differences between backup routers from different vendors in a university network. These were undetected for three years, and depended on subtle semantic differences that the operators said they were "highly unlikely" to detect by "just eyeballing the configs."

CCS Concepts

• **Networks** → **Network reliability**; **Network manageability**.

Keywords

Network Verification, Equivalence Checking, Error Localization, Modular Reasoning

ACM Reference Format:
Alan Tang, Siva Kesava Reddy Kakarla, Ryan Beckett, Ennan Zhai, Matt Brown, Todd Millstein, Yuval Tamir, and George Varghese. 2021. Campion: Debugging Router Configuration Differences. In *ACM SIGCOMM 2021 Conference (SIGCOMM '21), August 23–27, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3452296.3472925

1 Introduction

Networks today are manually configured through low-level configuration directives at individual routers that enforce complex policies for access control and routing. Manual programming often introduces subtle configuration errors that induce costly and disruptive outages [7, 19, 23, 25, 27, 30]. While researchers have developed many verification tools that can analyze network configurations to find bugs [1, 3, 4, 12, 13, 17, 18, 21, 24, 29, 32–34], there has been less focus on helping operators to understand and fix the identified bugs.

This paper presents an approach to router configuration *debugging* in the context of a specific, but common, verification task: *checking behavioral equivalence of two individual router configurations.* This task arises often in large networks. First, it is common for pairs of routers from different manufacturers (to avoid replicating implementation bugs) to serve as backups for one another in case of failure. Whenever one router in the pair is updated, the other must be consistently updated, which is non-trivial if they use different configuration formats. A second important use case is *router replacement.* As shown in (§ 5), routers are periodically upgraded from one manufacturer (e.g., Juniper) to one another (e.g., Arista) with better features, cost, or performance. Since the Arista configuration has to be manually translated from the Juniper, the operation is difficult and perilous. The first use case shows the need for behavioral equivalence checking in *space*, while the second is an example of the need for such checking in *time.*

Existing tools for network control-plane verification, such as Minesweeper [3], can be used to verify behavioral equivalence of two router configurations. However, while these tools can *detect* equivalence violations, they provide very little help in *debugging* such errors. In particular, existing tools have two key limitations that our work aims to address. First, they provide only a *single* counterexample and hence identify only a single behavioral difference between the two configurations. Second, the provided counterexample consists of a concrete packet whose forwarding exhibits a behavioral difference in the two configurations, leaving to the operator the difficult tasks of identifying the *set of packets* that is

SIGCOMM '21, August 23–27, 2021, Virtual Event, USA

ACM ISBN 978-1-4503-8383-7/21/08.
https://doi.org/10.1145/3452296.3472925

impacted and the specific *configuration lines* that caused the difference. We call the first challenge *header localization* and the second *text localization.*

We present a concrete example of header and text localization in §2. Figure 1 shows two example configuration snippets from real configurations for a Cisco and Juniper router, and Table 2 shows the differences output by our tool. The first few rows of each difference represent header localization and the last three rows represent text localization. While the configurations used in Figure 1 are small, they have subtle behavioral differences. Further, many enterprises have large route maps and ACLs of thousands of lines (see §5.1).

Our tool, Campion, performs localization through a novel *modular* approach. Rather than representing the behavior of each router configuration *monolithically*, for example as a set of SMT constraints [3], Campion compares pairs of corresponding components between the two configurations (route maps, ACLs, OSPF costs, etc., see Table 1) separately. Performing equivalence checks on a per-component basis immediately helps: every pair of components that are not behaviorally equivalent is reported, and each such violation is by construction localized to the relevant configuration components.

In the context of modular checking, two configuration components C_1 and C_2 are considered equivalent if *any* configuration containing C_1 could instead use C_2 without changing the configuration's behavior. How should each pair of components be checked for equivalence? Observe that there are two distinct types of configuration components from the point of view of modular checking.

Many configuration components have the property that any *structural* difference implies a possible behavioral difference. For example, two OSPF link costs are only guaranteed to be behaviorally equivalent, for all possible configurations, if they are identical. The same is true for static routes in two configurations. For these configuration components, we compare them with a simple *structural equivalence* check that we call StructuralDiff. This check is efficient, reports and localizes all behavioral differences — all structural mismatches — and makes it trivial for users to understand the error.

On the other hand, a few configuration components, specifically ACLs and route maps, encode sophisticated policies, so there are many possible structures for the same behavior, especially when considering multiple vendors. For example, Juniper and Cisco route maps are structured in very different ways. For these configuration components, we compare them with a *semantic equivalence* check that we call SemanticDiff. To identify *all* differences, we model the two components C_1 and C_2 as functions (e.g., an ACL is a function from a packet to a boolean). Then, for each path p_1 through C_1 and p_2 through C_2, we check whether there is some input that traverses along p_1 and p_2 through their respective components and exhibits a behavioral difference. This algorithm is conceptually similar to prior approaches to checking equivalence in C functions [26] and network data planes [9]. To our knowledge ours is the first approach that can precisely check equivalence of network control-plane structures, notably route maps.

The SemanticDiff algorithm localizes each behavioral difference to a specific path through each component. To help users understand the difference, we also introduce a novel algorithm called HeaderLocalize that localizes each difference to the relevant space of inputs. Specifically, SemanticDiff produces the impacted set of inputs I as a binary decision diagram (BDD). Given this BDD and the original configurations, HeaderLocalize produces a representation of all destination IP addresses in I in terms of the constants (prefixes or prefix ranges) that appear in the configurations, and does so in a minimal way.

Feature	Check Used
ACLs	SemanticDiff
Route Maps (BGP, Route Redistribution)	SemanticDiff
Static Routes	StructuralDiff
Connected Routes	StructuralDiff
Other BGP Properties	StructuralDiff
OSPF Properties (costs, areas, etc.)	StructuralDiff
Administrative Distances	StructuralDiff

Table 1: Components supported by Campion and the check used for each.

Perhaps surprisingly, Campion is *protocol-free*: it does not need to model or reason about routing protocols like BGP and OSPF. Our modular approach obviates the need for such reasoning, as equivalence of each corresponding pair of configuration components implies that those protocols will behave identically on the two routers. We formally prove this theorem, thereby justifying our approach. A potential downside of our modular approach is that it can produce false positives: it is possible for two configuration components to cause a behavioral difference for *some* configuration, and hence be flagged as erroneous by Campion, but still be behaviorally equivalent in the context of the two given router configurations. However, our experiments indicate that false positives are rare. Intuitively this makes sense because configurations are created and maintained in a modular fashion, with different aspects of the configuration responsible for different aspects of the behavior.

We evaluated Campion on the network configurations of a large cloud provider and a large university campus. We highlight two key results, with details in §5. First, the operators of the cloud provider were in the process of replacing 30 Cisco routers with Juniper routers due to a corporate policy decision. This required them to manually translate the original Cisco IOS configurations to JunOS. They used Campion to proactively check equivalence, identifying four configuration errors that they fixed before they could cause service disruption, including one error that would have been a severe outage. Second, the university network has a pair of core routers and a pair of border routers from different device vendors and intended to be backups of one another. Campion identified and localized configuration errors across these two pairs. These errors have been present in the configurations for nearly three years, and the operators said that they were "highly unlikely" to detect them by "just eyeballing the configs." Campion only takes a few seconds to compare a pair of routers. Our work does not raise any ethical issues.

To summarize, the contributions of this paper are:

- A modular approach that identifies *all* behavioral differences between two configurations and localizes them to the relevant configuration lines (§3). For each configuration component, we determine whether a full semantic analysis (SemanticDiff) is needed or a simple structural equivalence check (StructuralDiff) suffices (see Table 1). We also describe a novel algorithm for localizing the relevant inputs (HeaderLocalize).

- A theorem (§3.4) that shows our modular approach to equivalence checking of configuration components suffices to ensure router behavioral equivalence, despite not reasoning about the network protocols.
- A tool, Campion (§4), that localizes behavioral differences between router configurations. Campion supports all of the routing and forwarding components modeled by Minesweeper. Campion is available as open-source software.[1]
- An experimental evaluation of Campion on routers from a large cloud vendor and a university network. (§5).

2 Campion by Example

This section shows two examples of Campion's output that identified behavioral differences in routers from a large university network. We present one case involving differences between BGP route maps, which Campion identified and localized using SemanticDiff and HeaderLocalize, and a second case involving differences in static routes, which Campion identified and localized using StructuralDiff. In both cases, we also demonstrate the advantages of Campion by comparing its output to that of Minesweeper [3], a state-of-the-art network configuration verification tool.

2.1 Route Map Diffs via Semantic Checks

Figure 1 shows simplified versions of route maps from two core routers in a large university network (see § 5.2). The two route maps are intended to be behaviorally identical, with the first written for a Cisco router and the second for a Juniper router. Both configurations define a prefix list NETS to match a specific set of IP prefixes (lines 1-2 in Figure 1(a) and 1-4 in Figure 1(b)), as well as a community list COMM to match the community tags 10:10 and 10:11 (4-5 in Figure 1(a) and 5 in Figure 1(b)). The remainder of each snippet defines a route map POL for each router, which rejects route advertisements that match prefixes from NETS or are tagged with communities from COMM and accepts all other advertisements (7-12 in Figure 1(a) and 6-21 in Figure 1(b)).

Despite the superficial similarity of the two configurations, there are large behavioral differences. Campion uses SemanticDiff and HeaderLocalize to find and localize these differences. Table 2 shows Campion's output when given the two route maps in Figure 1. The output has two results, each of which represents a distinct configuration error. For each error, Campion identifies *all* the route advertisement prefixes that are treated differently by the two route maps, namely route advertisements for prefixes that are in the set Included Prefixes but not the set Excluded Prefixes. We call the process of identifying and representing all problematic inputs *header localization.* Further, Campion also shows the action that each route map takes on these advertisements as well as the configuration lines responsible for that action. We call the process of identifying all relevant lines of the configuration *text localization.*

In the output shown in Table 2(a), the Action and Text rows indicate that advertisements for the relevant prefixes match the NETS prefix list in the Cisco route map and are therefore rejected, but these prefixes fall through to the last term in the Juniper route map and are accepted. Careful inspection reveals the problem: in the Cisco route map, NETS matches prefixes with lengths between 16 and 32, while in the Juniper route map it only matches prefixes with lengths of exactly 16. Thus, a prefix like 10.9.1.0/24 is matched by the Cisco route map but not by the Juniper route map.

```
1   ip prefix-list NETS permit 10.9.0.0/16 le 32
2   ip prefix-list NETS permit 10.100.0.0/16 le 32
3   !
4   ip community-list standard COMM permit 10:10
5   ip community-list standard COMM permit 10:11
6   !
7   route-map POL deny 10
8    match ip address NETS
9   route-map POL deny 20
10   match community COMM
11  route-map POL permit 30
12   set local-preference 30
```

(a) Excerpt from the Cisco route map

```
1       prefix-list NETS {
2           10.9.0.0/16;
3           10.100.0.0/16;
4       }
5       community COMM members [10:10 10:11];
6       policy-statement POL {
7           term rule1 {
8               from prefix-list NETS;
9               then reject;
10          }
11          term rule2 {
12              from community COMM;
13              then reject;
14          }
15          term rule3 {
16              then {
17                  local-preference 30;
18                  accept;
19              }
20          }
21      }
```

(b) Excerpt from the Juniper route map

Figure 1: Cisco and Juniper route maps with subtle differences

The second result that Campion produces (Table 2(b)) identifies a second, unrelated configuration difference. The Included Prefixes and Excluded Prefixes rows show that this difference occurs for advertisements of all prefixes other than those in the ranges of the NETS prefix list. While Campion can find all differences and identify all relevant IP prefixes, for other fields of the route advertisement it currently provides a single example. In this case, the output indicates that this difference occurs when the route advertisement contains only the community 10:10. The Action and Text rows show that the Cisco route map matches the advertisement against the community list COMM and rejects it, while the Juniper route map again falls through to the last rule. This difference reveals a subtle error: COMM in the Cisco route map matches route advertisements containing *either* the community 10:10 or 10:11, whereas COMM in the Juniper route map erroneously matches only advertisements tagged with *both* communities.

Campus network operators confirmed both of the above behavioral differences as configuration errors. Further, the errors are subtle and have existed since at least July 2017. The network operator commented, "your config-analysis tool is great. It's highly unlikely anyone would detect the functional discrepancies just by

[1] https://github.com/atang42/batfish/tree/rm-localize

	cisco_router	juniper_router
Included Prefixes	10.9.0.0/16 : 16-32 10.100.0.0/16 : 16-32	
Excluded Prefixes	10.9.0.0/16 : 16-16 10.100.0.0/16 : 16-16	
Policy Name	POL	POL
Action	REJECT	SET LOCAL PREF 30 ACCEPT

Text	route-map POL deny 10
	 match ip address NETS
		rule3 {
		 then {
		  local-preference 30;
		  accept;
		 }
		}

(a) Difference 1

	cisco_router	juniper_router
Included Prefixes	0.0.0.0/0 : 0-32	
Excluded Prefixes	10.9.0.0/16 : 16-32 10.100.0.0/16 : 16-32	
Community	10:10	
Policy Name	POL	POL
Action	REJECT	SET LOCAL PREF 30 ACCEPT

Text	route-map POL deny 20
	 match community COMM
		rule3 {
		 then {
		  local-preference 30;
		  accept;
		 }
		}

(b) Difference 2

Table 2: Campion result when checking equivalence of configurations in Figure 1 using a Semantic Check

Route received (Cisco)	Prefix: 10.9.0.0/17
Route received (Juniper)	Prefix: 10.9.0.0/17
Packet	dstIp: 10.9.0.0
Forwarding	Juniper router forwards (BGP) Cisco router does not forward

Table 3: Minesweeper result when checking equivalence of configurations from Figure 1

eyeballing the configs." As described in §5.2, Campion found additional differences that have been removed here to keep the example simple.

Comparison with Minesweeper. Minesweeper [3] builds a logical representation of the network behavior, modeling the routing process and forwarding behavior. It then uses a satisfiability modulo theories (SMT) solver to answer verification queries. Minesweeper supports a behavioral equivalence check of individual routers, but it does so by checking that the logical representation of both routers' entire configurations are equivalent. A major drawback of this monolithic approach is the difficulty to diagnose the source of the error — any identified difference could be caused by BGP configuration, OSPF configuration, ACLs, or static routes.

	cisco_router	juniper_router
Prefix	10.1.1.2/31	
Next Hop	10.2.2.2	None
Admin. Distance	1	None
Text	ip route 10.1.1.2 255.255.255.254 10.2.2.2	None

Table 4: Campion result when checking equivalence of static routes using a Structural Check

In order to make the comparison more fair, we adapted Minesweeper to only check behavioral equivalence of two route maps. Specifically, Minesweeper checks that its logical representations of the two route maps are equivalent: whenever they receive the same set of route advertisements, they produce the same forwarding behavior for all packets. Table 3 shows the output of this modified version of Minesweeper on the above example. There is a single counterexample indicating that, when both routers receive a route advertisement with prefix 10.9.0.0/17, they will produce different rules for forwarding packets with destination IP address 10.9.0.0: the Juniper router will forward them, while the Cisco router will not.

Minesweeper's output identifies a behavioral difference between the two route maps that corresponds to Campion's output shown in Table 2(a). However, Minesweeper's output is lacking in several important ways. (1) It only provides information about a single behavioral difference. However, as explained earlier, there are actually two unrelated configuration differences between these route maps (Table 2(a) and Table 2(b)). (2) For the error that Minesweeper does identify, it only provides a single concrete example, with a specific route advertisement and destination IP prefix. To fully fix the problem of unintended differences between the two route maps, operators must understand the set of all route advertisements that produce this behavioral difference. Having this set explicitly also provides an indication of the scope of the problem. (3) Minesweeper provide no information about what parts of the route maps are responsible for the behavioral difference.

It is possible to modify Minesweeper again, this time to produce multiple concrete examples. This can be done by simply querying the SMT solver multiple times, each time including additional logical constraints that disallow previously generated counterexamples. This approach could potentially alleviate the first two issues described in the previous paragraph, but our experiments with this approach illustrate that it is not very effective. On the above example, running Minesweeper does provide counterexamples from both classes of differences from Table 2 but it takes 7 counterexamples in order to have at least one for each prefix range that is relevant for Difference 1. Further, the approach is fragile: when we replaced the number 32 in the second line of the Cisco configuration (Figure 1(a)) with 31, it took 27 counterexamples for Minesweeper to provide a violation of Difference 1 instead of Difference 2.

2.2 Static Route Diffs via Structural Checks

Campion detects differences in configuration components such as static routes and OSPF costs using a structural equivalence check. For example, for static routes Campion simply considers the set

Packet	dstIp: 10.1.1.2
Forwarding	Cisco router forwards (static) Juniper router does not forward

Table 5: Minesweeper result when checking equivalence of static routes

of static routes in each router and identifies all structural differences: cases where a route is present in one set but not the other, or where a route is present in both but with different attributes such as the next hop and administrative distance. This technique illustrates another advantage of our modular approach. Because we are checking configuration components in isolation from the rest of the configurations, for many components a simple structural check is *as precise as* a behavioral check via a semantic representation, while providing better localization and understandability for users.

An example of an output produced by Campion when checking static routes is shown in Table 4. This output shows that in the Cisco router, a static route exists that sends packets destined to `10.1.1.2/31` to `10.2.2.2`, but there is no such route in the Juniper router. Differences like this were found in both the university and cloud networks.

Table 5 shows the output that Minesweeper produces for the same example. Minesweeper can identify that the forwarding was caused by a static route, but it does not determine the prefix of the static route, the other relevant fields like the administrative distance, or the lines of the configuration. Hence operators have to search through a potentially large set of static routes and determine which one would affect the routing of packet to a `10.1.1.2`. Further, if there were multiple static-route differences, Minesweeper would only find one, while Campion would identify all.

3 Design and Algorithms

We describe Campion's design and core algorithms. Campion's overall algorithm for identifying and localizing behavioral differences between configurations C_1 and C_2 is as follows:

```
func ConfigDiff ($C_1, C_2$)
    result ← [ ]
    pairs ← MatchPolicies($C_1, C_2$)
    for $(p_1, p_2) \in$ pairs do
        differences ← Diff($p_1, p_2$)
        for $d \in$ differences do
            result ← result.append(Present($d$, $\{C_1, C_2\}$))
    return result
```

This algorithm consists of three main parts:

(1) The corresponding components (ACLs or BGP route maps) for C_1 and C_2 are paired up by the MatchPolicies function. This can be done with heuristics such as matching components by name or matching components that relate to the same neighboring node, or this information can be provided by the user.

(2) For each component pair, the Diff function invokes either SemanticDiff or StructuralDiff to produce a set of *differences*, each of which can include a set of inputs, the actions taken by each component, and the locations in the configurations.

(3) The Present function formats the results for output to the user, including invoking HeaderLocalize on the results of SemanticDiff in order to produce an understandable representation of the set of inputs.

We now describe SemanticDiff, HeaderLocalize, and StructuralDiff in more detail. We then discuss the general applicability of SemanticDiff and StructuralDiff and show how our modular approach can find and localize behavioral differences across entire router configurations.

3.1 SemanticDiff

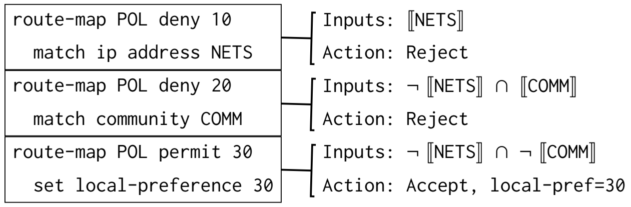
SemanticDiff takes a pair of configuration components as input and returns a list of all behavioral differences. The same basic algorithm applies to both ACLs and route maps. Each difference is a quintuple of the form: (i, a_1, a_2, t_1, t_2). In this quintuple, i refers to a set of inputs to the components, represented as a logical formula over message headers. For dataplane ACLs the inputs are sets of packets, and for route maps they are route advertisements. a_1 and a_2 are the respective actions taken by the two components when given an input from i. The action for ACLs is either accept or reject, but for route maps the accept action can also set fields such as local preference. t_1 and t_2 are the respective lines of text from the two components that process inputs from i and result in a_1 and a_2.

The SemanticDiff algorithm has two main steps. First, for each configuration component, the space of inputs is divided into equivalence classes, based on their *paths* through the component. Both ACLs and route maps can be viewed as a sequence of *if-then-else* statements, so two inputs are in the same equivalence class if and only if they take the same set of branches through these statements. Each equivalence class is represented symbolically as a logical predicate on the input (either a packet header or route advertisement). Our implementation uses BDDs to represent these predicates. Each equivalence class is also associated with the text lines that are on the corresponding path as well as the action taken. This step consequently produces two lists of triples:

$$L_1 = [(i_{1,1}, a_{1,1}, t_{1,1}),\ (i_{1,2}, a_{1,2}, t_{1,2}), \ldots,\ (i_{1,m}, a_{1,m}, t_{1,m})]$$
$$L_2 = [(i_{2,1}, a_{2,1}, t_{2,1}),\ (i_{2,2}, a_{2,2}, t_{2,2}), \ldots,\ (i_{2,m}, a_{2,m}, t_{2,m})]$$

Figure 2 shows the equivalence classes for the example route map from Figure 1(a). `NETS` and `COMM` correspond to the names of the attribute filters — `NETS` for prefix filters and `COMM` for communities. We use $[\![\texttt{NETS}]\!]$ to denote the set of accepted prefixes, and similarly $[\![\texttt{COMM}]\!]$ to denote the set of accepted communities. We also denote the complement of a set X as $\neg X$. There are three equivalence classes, one per clause in the route map — the first clause is associated with the space $[\![\texttt{NETS}]\!]$, the second clause is associated with $\neg [\![\texttt{NETS}]\!] \cap [\![\texttt{COMM}]\!]$, the space of routes matching $[\![\texttt{COMM}]\!]$ but not $[\![\texttt{NETS}]\!]$, and the third clause is for all remaining routes. Each equivalence class is also associated with whether it accepts or rejects routes and what fields are set.

Once the inputs are partitioned into equivalence classes for both components, the SemanticDiff algorithm then performs a pairwise comparison to identify behavioral differences. For each pair of equivalence classes $(i_{1,i}, a_{1,i}, t_{1,i})$ and $(i_{2,j}, a_{2,j}, t_{2,j})$ from the two components, if $i_{1,i}$ and $i_{2,j}$ have a non-empty intersection and the actions $a_{1,i}$ and $a_{2,j}$ differ, then there is a behavioral difference. In



```
route-map POL deny 10               Inputs: ⟦NETS⟧
  match ip address NETS             Action: Reject
route-map POL deny 20               Inputs: ¬ ⟦NETS⟧ ∩ ⟦COMM⟧
  match community COMM              Action: Reject
route-map POL permit 30             Inputs: ¬ ⟦NETS⟧ ∩ ¬ ⟦COMM⟧
  set local-preference 30           Action: Accept, local-pref=30
```

Figure 2: Partitioning the space of route advertisements based on route map definitions.

that case, we add

$$(i_{1,i} \cap i_{2,j},\ a_{1,i},\ a_{2,j},\ t_{1,i},\ t_{2,j})$$

to the list of differences returned by SemanticDiff.

3.2 HeaderLocalize

SemanticDiff produces the set of packets that exhibit behavioral differences as a logical predicate. The HeaderLocalize algorithm produces a more human-understandable representation in terms of the constants (e.g. IP prefixes) that appear in the configuration, handling the *header localization* problem. Specifically, HeaderLocalize produces a compact representation of the set of all destination IP addresses relevant to an ACL difference and the set of all IP prefix ranges relevant to a route map difference. For ease of presentation, we only describe finding prefix ranges relevant to route map differences, but the process for ACLs is analogous. In principle, HeaderLocalize can also be extended to other route fields such as communities, but we have not yet done so. Currently, instead of producing all communities relevant to a route map difference, Campion outputs a single example.

For route maps, sets of IP prefixes are represented by *prefix ranges*, each of which is a pair of a prefix and a range of lengths. For example, (`1.2.0.0/16, 16-32`) is a prefix range where the prefix is `1.2.0.0/16` and the length range is `16-32`. A prefix p is a member of a prefix range R if both of the following hold:

(1) The IP address of p matches the prefix of R
(2) The length of p is included inside the range of R

For example, `1.2.3.0/24` is a member of the prefix range (`1.2.0.0/16, 16-32`), (`0.0.0.0/0, 0-32`) is the set of all prefixes, and (`1.0.0.0/8, 24-24`) is the set of all prefixes with length 24 and 1 as the first octet. We say that a prefix range R_1 is contained in another prefix range R_2, denoted $R_1 \subset R_2$, if the members of R_1 are a subset of those of R_2.

The input to HeaderLocalize is a BDD S representing the set of messages affected by an identified policy difference, along with the original configurations C_1 and C_2. The output is a representation of S's prefix ranges in terms of the prefix ranges that are in the two configurations. First, all prefix ranges from the two configurations are extracted to get the set $\mathcal{R} = \{R_1, R_2, \ldots R_n\}$. If the set of all prefixes (`0.0.0.0/0, 0-32`), which we will call U, is not in $\mathcal{R}$, then it is added. Furthermore, $\mathcal{R}$ is extended so that it is closed under intersection. Since each line of a route map can allow or reject route advertisements based on prefix ranges in the configuration, it is always possible to represent the set S as a combination of complements, unions, and intersection of sets from $\mathcal{R}$. The goal of HeaderLocalize is to identify the minimal such representation.

To find this minimal representation, HeaderLocalize builds a directed acyclic graph (DAG) that relates the prefix ranges in $\mathcal{R}$ to one another. This data structure is analogous to the ddNF data structure previously used for packet header spaces [8], but here we associate each node with prefix ranges rather than tri-state bit vectors representing data-plane packets. HeaderLocalize's ddNF data structure consists of a set of nodes N, a set of edges $E \subseteq N \times N$, a labeling function l mapping nodes to prefix ranges, and a root node. It satisfies the following properties:

(1) The root node is labeled with U, the set of all prefixes, and all other nodes are reachable from it.
(2) Each node has a unique label (and thus in the following explanation, we will sometimes refer to a node by its prefix range or vice versa).
(3) The set of prefix ranges used as labels contains $\mathcal{R}$ and is closed under intersection.
(4) For any nodes $m, n \in N$, there is an edge $(m, n) \in E$ exactly when $l(n) \subset l(m)$ and there is no node m' such that $l(n) \subset l(m') \subset l(m)$.

An example DAG is shown in Figure 3 for a set of seven prefix ranges. There is one node per prefix range, and each node's prefix range is a subset of those of its ancestors. For example D is contained in B and A. The DAG is built by inserting one prefix range at a time, starting with U [8]. We also associate each internal node, with prefix range R and outgoing edges to nodes labeled $C_1, C_2, \ldots, C_k$, with the set of prefixes $R - C_1 - C_2 \cdots - C_k$. We call this set the *remainder* set, as it is the set of prefixes that remain in R after prefixes of the children nodes are removed. For example, the remainder set of node B in Figure 3 is $B - D - E$. The remainder and leaf node sets are all disjoint from one another, and their union is U. Importantly, because the set S of interest was created through unions, intersections, and complements of the prefix ranges in $\mathcal{R}$, each remainder set and leaf prefix range has the property that either it is contained in S or disjoint from S.

Next HeaderLocalize uses the DAG to produce a representation of S in terms of the prefix ranges in $\mathcal{R}$. This is done by traversing the DAG with the recursive function GetMatch shown below. If the current node is a leaf, then its prefix range R is included in the result if that range is contained in S. If the current node is internal, then there are two cases. If the node's remainder is contained in S, then its prefix range R should be included in the result, after removing any of the node's child prefixes in the DAG that are not contained in S. This latter process is done through a recursive call to GetMatch with the complement set of S. if the node's remainder is not contained in S, then we simply recurse on the children and union the results.

The GetMatch algorithm produces a representation of S that is a union of terms of the form $R - X_1 - X_2 - \ldots X_k$, where R is a prefix range, but each X_j is also in the form $R - X_1 - X_2 - \ldots X_k$. For example, running GetMatch on the DAG in Figure 3 produces $\{B - D, C - (F - G)\}$, and the nodes in the figure are colored to illustrate the algorithm's process. As a final simplification step, we remove all *nested differences* from the result through a single pass over it. In our example, the result $C - (F - G)$ is transformed into $\{C - F, G\}$, so the final representation of the set S is $\{B - D, C - F, G\}$.

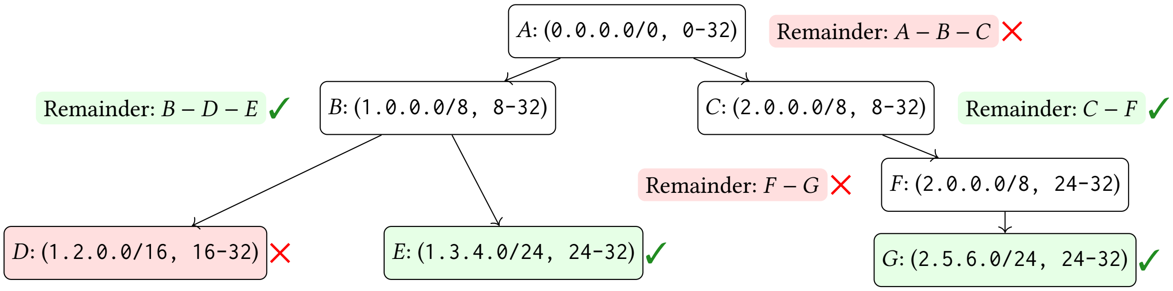


Figure 3: DAG created from prefix ranges. Green (✓) nodes represent leaves or remainders contained in a set S, and red (✗) nodes represent those that are not. S can be represented by the union of $B - D$, $C - F$, and G

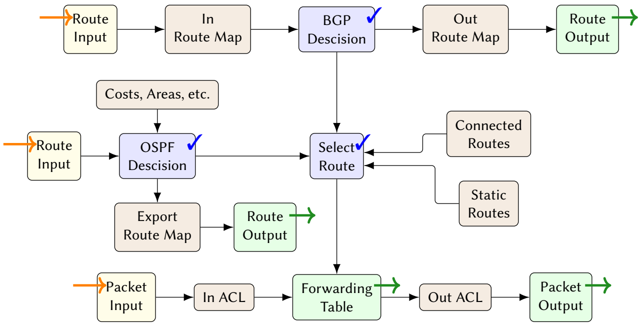


Figure 4: Basic features of routing and forwarding. Blue nodes(✓) represent fixed processes. Yellow nodes (incoming →) are inputs and green nodes (outgoing →) are outputs. Unmarked (brown) nodes represent configurable entities.

```
func GetMatch (S, node)
    C ⟵ Children(node)
    R ⟵ PrefixRange(node)
    if IsLeaf(node) then
        if R ⊆ S then
            return {R}                          ▶ node is a leaf, and $R \subseteq S$
        else
            return ∅                            ▶ node is a leaf, and $R \cap S = \emptyset$
    if Remainder(node, C) ⊆ $S$ then            ▶ checks if $R - C_1 - C_2 \ldots C_k \subseteq S$
        nonmatches ⟵ $\bigcup_{k \in C}$ GetMatch(¬ S, k)
        return {$R$ − nonmatches}               ▶ returns $\{R - X_1 - X_2 \ldots X_m\}$
    else
        return $\bigcup_{k \in C}$ GetMatch($S$, k)   ▶ returns $\{X_1, X_2 \ldots X_n\}$
```

3.3 StructuralDiff

It would be possible to use a semantic approach like SemanticDiff to reason about all configuration components, just as we do for route maps and ACLs. However, we observe that other configuration components typically have a very stylized structure, as a single atomic value (e.g., integer or boolean) or a simple collection of such values. Hence, when considered modularly, the equivalence of two such components is tied to their structure.

That is, two components are behaviorally equivalent, for all possible configurations, if and only if their structural representation is identical. Thus we can use a simple structural check without incurring additional false positives versus a semantic approach. Since the structural approach does not require logical modeling, it is more efficient. Further, localization is trivial since the structural check directly identifies the portions of the two components that differ.

Our StructuralDiff function implements this approach. All components are represented as atomic values, tuples, or unordered sets. Atomic values are tested for equality. Tuples are compared by testing that the corresponding values are equal. Finally, sets are compared using set difference.

For example, to check two OSPF configurations are equivalent (excluding route redistribution which is handled by SemanticDiff), it suffices to check equivalence for all corresponding attributes on all corresponding links. That means both routers must have OSPF edges to the same peers, and the corresponding edges are configured with the same costs, areas, passive status, etc. We can think of the configuration of each OSPF link as a tuple of its configured attributes and check each corresponding attribute. The same approach works for BGP properties not implemented with route maps, such as which edges are to route reflector clients and whether communities are propagated.

Other components that affect routing include connected and static routes. Connected routes are formed by the set of subnets connected to the router's interfaces, and the difference between routers is the set of such subnets present in one router but not the other. Similarly, a single static route can be represented as a tuple consisting of a destination prefix, a next-hop, an administrative distance, and optional fields like tags; so the difference is the set of tuples present in one router but not the other. Administrative distances can also be compared as values configured per protocol.

As mentioned earlier, localization for these components is straightforward because the equivalence check is performed directly on the components' structures. Further, unlike route maps and ACLs, these components have no explicit notion of input. Hence there is no need for, or analogue to, HeaderLocalize for such differences.

3.4 Debugging an Entire Router

We now formalize our approach to checking full router equivalence. We observe that many crucial parts of routing, such as the route selection process, are fixed. They are implemented according to a standard and depend only on the provided inputs and configurations. All of the various processes in Figure 4 need to be modeled to fully simulate a router or network, but only the configured aspects (shown in brown) need to be modeled to find behavioral differences.

Figure 4 provides a flow diagram illustrating the processes supported by Campion. For routing, there is both a BGP process (top of figure) and an OSPF process (middle of figure), as these are the most common inter-domain and intra-domain routing protocols; other protocols could be added similarly. The bottom of the figure shows the router's process for forwarding routes. The brown (unmarked) nodes represent parts of the router configuration, while the other components are fixed processes like routing protocols (in blue (✓)), or input routes and packets (in yellow (incoming →)), or outputs and byproducts like selected routes and forwarded packets (in green (outgoing →)).

Assuming that these are the only routing components used in the configurations being compared, then Campion is a sound verifier for router configuration equivalence: If Campion identifies no differences, then the two router configurations are behaviorally equivalent. We formalize the fact that behavioral equivalence can be verified without reasoning about the routing protocols as follows (our formalization considers behavioral equivalence of entire networks, but it therefore also applies to the special case of individual routers).

Definition 3.1. A network $\mathcal{N} = (T, \mathcal{R}, C_{\mathcal{P}}, \mathcal{F}_{\mathcal{P}}, \preceq_{\mathcal{P}})$ is a topology $\mathcal{T} = (\mathcal{V}, \mathcal{E})$ of vertices and edges, a set of routes $\mathcal{R}$, a family of configuration functions $C_{\mathcal{P}} : \mathcal{E} \rightarrow \Omega$ that maps each edge in the topology to a configuration Ω, a family of transfer functions $\mathcal{F}_{\mathcal{P}} : \Omega \times \mathcal{E} \times \mathcal{R} \rightarrow \mathcal{R}$ that transforms a route along an edge for a protocol, and a protocol preference relation $\preceq_{\mathcal{P}}: \mathcal{R} \times \mathcal{R}$ that compares two routes for a protocol.

Definition 3.2. For two networks $\mathcal{N} = (T, \mathcal{R}, C_{\mathcal{P}}, \mathcal{F}_{\mathcal{P}}, \preceq_{\mathcal{P}})$ and $\mathcal{N}^* = (T^*, \mathcal{R}, C^*_{\mathcal{P}}, \mathcal{F}^*_{\mathcal{P}}, \preceq_{\mathcal{P}})$ and an isomorphism $\mathcal{I}$ between $\mathcal{T}$ and $\mathcal{T}^*$, we say that the two networks are locally equivalent if for all protocols $p \in \mathcal{P}$, edges $e \in \mathcal{E}$, and routes $r \in \mathcal{R}$ then $\mathcal{F}_p(C_p(e), e, r) = \mathcal{F}^*_p(C^*_p(\mathcal{I}(e)), \mathcal{I}(e), r)$.

Theorem 3.3 (Soundness). *If networks $\mathcal{N}$ and $\mathcal{N}^*$ are locally equivalent for isomorphism $\mathcal{I}$, then they have the same set of routing solutions.*

Proof. The proof is by a reduction to the stable routing problem [4] and is described in the appendix. □

4 Implementation and Limitations

Campion operates on a vendor-independent representation produced by Batfish [12]. Real routers support an enormous number of features. For Campion, we have focused on the most common components used for routing and forwarding. Campion currently supports all of the configuration components and features that are supported by Minesweeper (Table 1). This includes common features of BGP route maps, like communities, local preference, and MEDs, as well as other configurable aspects of BGP like route redistribution. It also includes configurable OSPF attributes like link cost and areas, static routes, and ACLs. Sets of packets and route advertisements are represented by BDDs that are handled with the JavaBDD library, extending code from Bonsai [4] used to encode import filters, output filters, and ACLs.

As mentioned in the previous section, it is sometimes necessary to match up corresponding components between two routers. We used a few simple heuristics instead of manually specifying matching components. For BGP properties and route maps, we match up connections with the same neighbor id, and we report the neighbors that occur in one router but not the other. We match ACLs with the same name. For OSPF attributes, we match interfaces using a combination of their interface names, Batfish's inferred topology, and their IP address masks. This is necessary since interfaces in backup routers usually have different IP addresses. While these heuristics are not perfect, they allow Campion to be run quickly and easily.

Campion can identify differences and perform header localization for any vendor format that Batfish supports. However, currently Campion can only output exact text lines for configurations in Cisco IOS and Juniper JunOS formats, since we must write *unparsers* to convert Batfish's representation back to the original configuration text. For other formats, Campion does not produce exact text lines, but it still provides substantial localization information, including the component name, affected headers, and actions. Similarly, for some formats we do not show the exact text lines for StructuralDiff results, for example OSPF costs. But in these cases the localization information that Campion provides typically allows operators to find the relevant lines with simple text searches.

HeaderLocalize for route maps currently only provides exhaustive information for IP prefix ranges. For other relevant parts of a route advertisement such as community tags, Campion provides a single example. It is possible to extend HeaderLocalize to provide exhaustive information across multiple parts of a route advertisement, but doing so increases the complexity both of the algorithm and of its output. The current approach has been sufficient for operators to understand Campion's results and localize the errors.

5 Evaluation

We applied Campion to debug router configuration differences from a large cloud provider and the campus network of a large university, both of which employ a diversity of hardware router vendors. Our experiments demonstrate Campion's ability to identify cross-vendor configuration differences and to provide actionable localization information to operators.

5.1 Differencing in a large Data Center

Network *A* is from a global cloud vendor that uses routers from different manufacturers. We tested Campion on a data center network from vendor *A* that employs a Clos topology with hundreds of routers and thousands of servers. All routers are either Juniper or Cisco, whose configuration languages are supported by Campion. The data center network uses eBGP, iBGP, OSPF, static routes, ACLs, and route redistribution for the layer-3 routing topology. It carries business traffic for multiple global services. Each router configuration is thousands of lines.

Scenarios. We asked the network operators to employ Campion on three frequent, real and challenging tasks:

Scenario 1: Debugging redundant routers. Some routers (e.g., Top-of-Rack) are configured to be *backups* of one another with equivalent *modular* policies handling BGP, OSPF and static routes. For diversity, the operators deploy redundant routers from different vendors (e.g., Juniper, Cisco). Because network *A* took months to build, its current configuration comprises fragments written by

Scenario	Component	Structural or Semantic	Differences
Scenario 1	BGP	Semantic	5
	Static Routes	Structural	2
Scenario 2	BGP	Semantic	4
Scenario 3	ACLs	Semantic	3

Table 6: Data Center Network Results

different operators for diverse purposes, making hidden inconsistencies likely. It is important to not only ensure equivalence of multi-vendor, redundant routers, but also to *quickly* localize the root causes of any errors. Network *A* is constantly being reconfigured as more policies are added for upcoming production traffic. Campion allows greater agility by allowing new policies to be more quickly deployed in diverse backup routers. The operators used Campion to compare all pairs of backup routers.

Scenario 2: Router replacement. Network *A* has an important update called *router replacement*, where operators replace a router from one vendor with one from a different vendor. Such replacements occur several times a month to take advantage of the price, performance, and newer features. For example, the operators of network *A* might replace lower-version Cisco routers with higher-version Juniper routers in order to avoid a Cisco bug. Router replacement is one of the riskiest update operations in network *A*, since operators must manually rewrite the old configurations to the new format; many critical errors have occurred as a result. The operators used Campion to check for differences between old and new configurations before performing a scheduled replacement, in order to *proactively* detect errors.

Scenario 3: Access control in gateway routers. In network *A*, many ACL rules are applied in gateway routers for traffic control. All of network A's gateway routers should have identical access-control policies, but it is difficult for network A's operators to guarantee this since: (1) the number of ACL rules is very large, and (2) the use of nested ACL rules makes their logic complex. The operators used Campion to check the equivalence of ACL rules in the gateway routers of the data center network.

Output evaluation. Note that network *A*'s operators used Campion and its user interface *without any feedback or help from us in interpreting results.* The operators gave us very positive feedback on the practicality and usability of Campion. By using Campion, they found several risky, hidden configuration errors, as summarized in Table 6. All differences that Campion found were unintentional and considered to be errors by the operators. The network configurations had recently undergone a standardization process to replace ambiguous and "uncommonly-used" configuration commands with unambiguous and standard ones. Hence any differences found by Campion were likely to be erroneous, and indeed this was borne out by the lack of any false positives.

Scenario 1: Debugging redundant routers. Campion detected seven configuration bugs across all of the redundant router pairs that it analyzed. Five of the bugs represent missing fragments of BGP policy, and two of them were incorrect next hops in static routes. For four BGP bugs, Campion was able to accurately localize the difference. For example, Campion pointed out that a prefix for an import filter was missing in the primary router but present in the backup one. Why were these bugs not detected by customers or real-time monitoring systems? This was because the missing prefixes had not been used for production traffic yet, but would have been in the near future. Once a service using this prefix is enabled, a service problem would have occurred. Thus, Campion proactively prevented a future service disruption.

The fifth BGP error that Campion detected used a version of the Cisco IOS format which Campion does not fully support yet. Campion still detected the error and produced useful localization information, such as the relevant input space and the actions taken by each router, but the output configuration text was inaccurate. Due to this inaccuracy, the operator reported the need to spend more time to understand the precise bug location, but they still said that it was easy to spot the deviant configuration lines from Campion's output.

The two static route errors Campion detected were misconfigured next hops. Backup routers in network *A* should forward the same prefix to the same next hop, but Campion detected that they were configured to forward a particular prefix p to different next hops. This is very dangerous: a cascading failure would have triggered when the production traffic corresponding to p is turned on in the near future. Campion accurately pointed out non-equivalent next hops of this kind in *two* pairs of backup routers.

Scenario 2: Router replacement. We used Campion to test more than 30 router replacements. Campion successfully detected four bugs: one was an incorrect community number and three were incorrect local preferences. One local preference bug was for the replacement of a reflector device for iBGP. If this bug were not detected, the proposed replacement would have caused a severe outage.

Further, network *A*'s operators also tested Campion on a synthetic case based on a static route replacement which resulted in a significant outage one year ago. The tags of two static routes were configured differently due to a misunderstanding of the semantics of the two vendors. Campion accurately pointed out the difference between the static routes. In other words, a significant outage could have been avoided if Campion had been used a year ago.

Scenario 3: Access control in gateway routers. Campion successfully detected three ACL differences between gateway routers from Cisco and Juniper. Table 7 shows Campion's output for one of these differences.[2] Campion's text localization identified the exact line in the Cisco ACL where traffic was rejected. The Juniper ACL equivalent is divided into terms, and Campion's text localization was able to locate which term accepted the traffic. Further, Campion's header localization also identified header information like the relevant source IP prefix.

Running Time. For each of the above three scenarios, although the configuration files of each device in network *A* contains thousands of lines, Campion finished its localization task within five seconds for each pair of routers.

Comparing Campion with an existing tool. While provider *A* has its own home-grown verification system that has been used

[2] The IP addresses and ACL name in this figure have been anonymized for confidentiality reasons.

Router Pair	Route Map	Outputted Differences	Differences Reported	Confirmed	Pending
Core Routers	Export 1	5	5	4	1
	Export 2	1	1	1	0
Border Routers	Export 3	1	1	1	0
	Export 4	1	1	1	0
	Export 5	2	1	1	0
	Import	0	-	-	-

(a) SemanticDiff results on route maps

Router Pair	Component	Classes of Errors	Differences Reported	Confirmed	Pending
Core Routers	Static Routes	2	1	0	0
	BGP Properties	1	1	0	0

(b) StructuralDiff results

Table 8: University Network Results

	Router 1 (current)	Router 2 (reference)
Included Packets	srcIP: `9.140.0.3/32` dstIP: `0.0.0.0/0`	
Excluded Packets	srcIP: `9.140.0.3/32` dstIP: `0.0.0.0/0` protocol: `ICMP` +28 more	
ACL Name	`VM_FILTER_1`	`VM_FILTER_1`
Action	`REJECT`	`ACCEPT`
Text	`2299 deny ipv4 9.140.0.0 0.0.1.255 any`	`set firewall family inet filter VM_FILTER term permit_whitelist`

Table 7: An example for ACL rules debugging. Router 1 and Router 2 are Cisco and Juniper routers, respectively.

for 1.5 years, this system can only tell whether the network configuration meets operator intent, but does not provide any error localization capability. Thus, network A's operators spend considerable time localizing bugs even when the existing tool identifies bugs in the network. Campion therefore provides a new capability that can potentially reduce debugging time considerably for network A's operators.

Localization efficiency. For the configurations checked, all localization results were less than five lines of configuration code. The configuration files tested vary in size from 300 lines to more than 1000 lines. Of these, the number of lines that are part of an ACL or route map definition is typically more than 100. Campion thus drastically reduces the amount of configuration that operators must search through to debug a difference.

5.2 Differencing in a University Network

The university network consists of approximately 1400 devices, including border routers that connect to external ISPs, backbone core routers and building routers.

We ran Campion to compare the policies for a pair of core routers and a pair of border routers. In each pair, one used Cisco configuration format and the other used Juniper format. We chose these two pairs because they are the only Cisco-Juniper backup pairs with routing policy. The Cisco configurations and the Juniper core router configuration contain about 1800 lines of text. The Juniper border router configuration contains about 3500 lines of text. The results are shown in Table 8.

We match route maps that are applied to the same BGP neighbor. In total, there were five pairs of operator-defined export route maps, and one pair of operator-defined import route maps. The differences that Campion found are summarized in Table 8(a).

The prefix ranges, communities, and text lines produced by Campion made it straightforward to identify these discrepancies. The list of issues that we sent to the operators does not exactly correspond to the raw output of our tool. For example, since Campion divides sets of advertisements based on which lines process them, it is possible that a single underlying difference in the configuration results in multiple lines of outputted differences. In Table 8(a), the Outputted Difference column reports the number of raw outputs produced by Campion, whereas the Differences Reported column reports how many distinct issues we reported to the operators. We categorize a reported difference as Confirmed if the operator indicated that the identified difference was both an actual difference and unintentional. The last column indicates the number of reported differences whose status is unknown at this time.

As shown in the table, the operators confirmed that most of the differences Campion identified were in fact errors. Based on earlier snapshots, the differences have been present since at least July 2017.

The route maps shown earlier in Figure 1 illustrate two issues from a pair of core-router route maps (labeled Export 1 in Table 8(a)). These were differences in the definitions of a prefix list and a community set and were confirmed as unintentional discrepancies. For the difference in the prefix lists, the operator agreed it was a misconfiguration, but was not sure whether the Cisco or Juniper router was correct. For the community difference, the operator wrote: "The community group is an obvious mistake on our part. The Juniper config is wrong. We followed the wrong Juniper doc when configuring the community group."

In addition to the differences shown in Figure 1, the actual route maps contained different definitions for their third clause, with the Juniper router performing a match on communities that was not done in the Cisco router. They also have different redistribution behavior for certain addresses. Further, the two routers have different fall-through behaviors (accept vs. deny) when handling advertisements that fail to match any clause, which causes two additional behavioral differences. Operators confirmed all but the last of these issues, which is still pending. When asked about the difference between the third clauses of each route map, the operator replied: "The Juniper config is correct and the intent is obvious because of the English-language syntax. The Cisco config we're not sure what change should be made, if any." This demonstrates the challenge for operators when dealing with multi-vendor backups, and the need for a tool like Campion to ensure consistency and localize errors.

Export 2, the other core router policy, also had the difference in prefix lists mentioned previously for Export 1 but did not have any other issues. The differences in the border router policies similarly affected the matched prefixes and communities but were of a different nature: there were differences in two regular expressions used to match communities for Export 3 and Export 4. Campion reported that advertisements with a certain community were accepted in the Cisco router but not the Juniper router. For Export 5, there was one prefix that was absent in a prefix list in the Juniper router but present in the Cisco router list. These were also confirmed as errors by the operators.

When comparing other properties of the core routers using Campion's StructuralDiff, we found differences in the static route configuration and the BGP configuration. In the static routes we found two classes of differences. The first included many static routes that applied to the same prefix but had different next hops and different administrative distances. We deemed these as intentional differences, since the next hops had similar addresses, suggesting that their next hop routers were of the same role, and the administrative distances did not affect the relative priority of routes. The second class of static route differences included two static routes that were present in one router but not the other, as demonstrated in § 2. These were reported to the operators, and they said that these were intentionally added as a workaround for a specific BGP routing issue. The BGP configuration difference was that certain iBGP neighbors of the Cisco router were missing a `neighbor send-community` command to propagates communities, while Juniper routers send communities by default. The operators indicated that this configuration difference does not cause a behavioral difference because the core routers do not set communities on routes.

5.3 False Positives

We distinguish between two types of false positives that Campion may produce, both of which were exhibited in the results for the university network. First, there can be intentional differences between routers. This was the situation for the static routes that were added in one configuration as a workaround for a specific BGP routing issue, as well as for the static routes that had differing next hops. Second, there can be spurious differences due to Campion's modular approach. Specifically, any *potential* behavioral difference between corresponding components is reported by Campion, but these differences may not cause an *actual* behavioral difference in the current network, for example because the differences are shadowed or accounted for by other parts of the configuration. This was the situation for the iBGP neighbors of one router which were not configured to send communities.

However, we argue that it is still worthwhile to report both kinds of false positives. Reporting intentional differences allows the operator to ensure that all and only expected differences exist between the two routers. In the case of static routes added as a workaround, the operator commented, "I just need to find another way to resolve this," indicating that this difference is intentional but still not optimal. Reporting spurious differences is valuable because they represent *latent* errors that can potentially be "activated" by a change elsewhere in the network configuration. In the case of the spurious difference for sending communities, if the core routers later start to set communities on routes then this difference will cause an important behavioral difference. Indeed, the operator commented that these kinds of spurious differences would likely be examined and addressed when the routers are next replaced.

5.4 Scalability

For each of the data center scenarios, Campion finished its localization task within five seconds for each pair of routers. For the university core and border pairs, the total runtime to compare the core and border pairs was 3 seconds. When combined with the parsing of the configurations, the total time was under 10 seconds, with configuration parsing taking a majority of the time. We additionally tested the scalability of SemanticDiff for ACLs. We used Capirca[3] to randomly generate nearly equivalent ACLs for Cisco and Juniper configurations. We introduced 10 differences between the two ACLs and compared them. When the ACLs were generated with 1000 rules, SemanticDiff took less than a second. When the ACLs were generated with 10,000 rules, SemanticDiff took 15 seconds. These tests were done with a 2.2 GHz CPU. Moreover, Batfish's parsing time for the 10,000 case is 13 seconds, which is comparable to the runtime of SemanticDiff.

6 Related Work

At a high level our work differs from prior work in network verification in two ways. First, we target verifying behavioral equivalence of two router configurations, while prior work typically targets network-wide reachability properties. Second, we localize identified errors to both relevant headers and configuration lines; most prior work simply provides individual concrete counterexamples.

Data Plane Verification Tools: Many tools verify reachability properties of a network's data plane, including its ACLs and forwarding tables [2, 15, 17, 18, 20, 21, 32]. Several tools focus on ACLs [22, 29] and localize errors to ACL lines [14, 15, 17, 29]. Closest to our work, `netdiff` [9] is a tool for checking data plane equivalence in networks using a similar symbolic execution approach, but it focuses on the data plane. Campion extends these capabilities to perform configuration localization for the control plane. HeaderLocalize and StructuralDiff have no analogue in `netdiff`.

Control Plane Verification: Other tools verify properties of a network's control plane routing processes [1, 3, 4, 10, 12, 24, 31, 33]. These tools can be adapted to perform router equivalence checking, as we showed for Minesweeper [3] in § 2. However, when verification fails, these tools only provide individual, concrete counterexamples, while Campion localizes to both headers and configuration text. As we have seen by the experiment in Section 2, even if we extend Minesweeper to produce multiple counterexamples it is still not able to quickly find all errors. Further, this still leaves the question as to which parts of the text caused each error. Recent work extends Minesweeper to localize errors by leveraging an SMT solver's ability to provide *unsatisfiable cores* when verification fails [28]. The approach localizes errors to specific SMT constraints, but not to configuration lines or headers. Campion leverages the BDD encoding of ACLs and route maps from Bonsai [4], which uses BDDs to perform network abstraction, not router differencing or debugging. Campion's structural checks are reminiscent of `rcc` [11], but our checks are designed to ensure behavioral equivalence and to do so without incurring additional false positives over a modular semantic check.

Outlier Detection: Benson *et al.* [5, 6] infer data-plane reachability specifications from a network's forwarding tables and use these specifications in part to identify outliers. However, they only consider the data plane and cannot localize back to the original

[3] https://github.com/google/capirca

configurations. SelfStarter [16] infers parameterized configuration templates for ACLs and route maps and uses them for outlier detection. This approach uses sequence alignment and so requires router configurations to be structurally similar. Further, SelfStarter localizes configuration text but cannot localize headers.

Equivalence Checking: Equivalence checking is an old idea beyond networks, and our SemanticDiff algorithm is similar in spirit to prior work. For example, Ramos *et al.* [26] perform equivalence checking of two C functions via pairwise comparisons of execution paths. Because network ACLs and route maps are loop-free, Campion is exhaustive, finding *all* differences and localizing to *all* IP prefixes; equivalence checking of software is undecidable in general.

7 Conclusion

Campion is a tool for debugging router configurations *intended* to be behaviorally equivalent but which in fact are not. Unlike prior work, Campion uses modular structural or semantic checks to localize errors to the affected message headers and relevant configuration lines. Our experience with a cloud provider and a university indicates that Campion satisfies a real need by localizing crucial errors.

Prior control-plane verification tools model a configuration monolithically as a set of constraints. In contrast, Campion exploits the modular structure of configurations to break up complex checks of whole router behavior into smaller per-component checks. This "bottom up" style eases localization, sidesteps reasoning about the routing protocols, and allows simple structural checks to often be used without additional loss of precision. None of these capabilities would be possible without exploiting modularity. As in other forms of verification, we believe exploiting modularity will be critical to making real-world network verification and debugging effective.

Acknowledgments

Thanks to the SIGCOMM reviewers for their helpful comments. Thanks to the network operators for using Campion and providing feedback on its results. This work was supported in part by NSF grants CNS-1704336 and CNS-1901510.

References

[1] Anubhavnidhi Abhashkumar, Aaron Gember-Jacobson, and Aditya Akella. 2020. Tiramisu: Fast Multilayer Network Verification. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 201–219. https://www.usenix.org/conference/nsdi20/presentation/abhashkumar

[2] Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. 2014. NetKAT: Semantic Foundations for Networks. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL '14)*. Association for Computing Machinery, New York, NY, USA, 113–126. https://doi.org/10.1145/2535838.2535862

[3] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. 2017. A General Approach to Network Configuration Verification. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication* (Los Angeles, CA, USA) *(SIGCOMM '17)*. Association for Computing Machinery, New York, NY, USA, 155–168. https://doi.org/10.1145/3098822.3098834

[4] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. 2018. Control Plane Compression. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication* (Budapest, Hungary) *(SIGCOMM '18)*. Association for Computing Machinery, New York, NY, USA, 476–489. https://doi.org/10.1145/3230543.3230583

[5] Theophilus Benson, Aditya Akella, and David Maltz. 2009. Unraveling the Complexity of Network Management. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation* (Boston, Massachusetts) *(NSDI'09)*. USENIX Association, Berkeley, CA, USA, 335–348. http://dl.acm.org/citation.cfm?id=1558977.1559000

[6] Theophilus Benson, Aditya Akella, and David A. Maltz. 2009. Mining Policies from Enterprise Network Configuration. In *Proceedings of the 9th ACM SIGCOMM Conference on Internet Measurement* (Chicago, Illinois, USA) *(IMC '09)*. ACM, New York, NY, USA, 136–142. https://doi.org/10.1145/1644893.1644909

[7] TODD Bishop. 2013. Xbox Live outage caused by network configuration problem. https://www.geekwire.com/2013/xbox-live-outage-caused-network-configuration-problem/

[8] Nikolaj Bjørner, Garvit Juniwal, Ratul Mahajan, Sanjit A. Seshia, and George Varghese. 2016. ddNF: An Efficient Data Structure for Header Spaces. In *Hardware and Software: Verification and Testing*, Roderick Bloem and Eli Arbel (Eds.). Springer International Publishing, Cham, 49–64.

[9] Dragos Dumitrescu, Radu Stoenescu, Matei Popovici, Lorina Negreanu, and Costin Raiciu. 2019. Dataplane equivalence and its applications. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 683–698. https://www.usenix.org/conference/nsdi19/presentation/dumitrescu

[10] Seyed K. Fayaz, Tushar Sharma, Ari Fogel, Ratul Mahajan, Todd Millstein, Vyas Sekar, and George Varghese. 2016. Efficient Network Reachability Analysis Using a Succinct Control Plane Representation. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Savannah, GA, USA) *(OSDI'16)*. USENIX Association, USA, 217–232.

[11] Nick Feamster and Hari Balakrishnan. 2005. Detecting BGP Configuration Faults with Static Analysis. In *Proceedings of the 2Nd Conference on Symposium on Networked Systems Design & Implementation - Volume 2 (NSDI'05)*. USENIX Association, Berkeley, CA, USA, 43–56. http://dl.acm.org/citation.cfm?id=1251203.1251207

[12] Ari Fogel, Stanley Fung, Luis Pedrosa, Meg Walraed-Sullivan, Ramesh Govindan, Ratul Mahajan, and Todd Millstein. 2015. A General Approach to Network Configuration Analysis. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. USENIX Association, Oakland, CA, 469–483. https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/fogel

[13] Aaron Gember-Jacobson, Raajay Viswanathan, Aditya Akella, and Ratul Mahajan. 2016. Fast Control Plane Analysis Using an Abstract Representation. In *Proceedings of the 2016 ACM SIGCOMM Conference* (Florianopolis, Brazil) *(SIGCOMM '16)*. ACM, New York, NY, USA, 300–313. https://doi.org/10.1145/2934872.2934876

[14] Karthick Jayaraman, Nikolaj Bjørner, Jitu Padhye, Amar Agrawal, Ashish Bhargava, Paul-Andre C Bissonnette, Shane Foster, Andrew Helwer, Mark Kasten, Ivan Lee, Anup Namdhari, Haseeb Niaz, Aniruddha Parkhi, Hanukumar Pinnamraju, Adrian Power, Neha Milind Raje, and Parag Sharma. 2019. Validating Datacenters at Scale. In *Proceedings of the ACM Special Interest Group on Data Communication* (Beijing, China) *(SIGCOMM '19)*. Association for Computing Machinery, New York, NY, USA, 200–213. https://doi.org/10.1145/3341302.3342094

[15] Karthick Jayaraman, Nikolaj Bjørner, Geoff Outhred, and Charlie Kaufman. 2014. *Automated Analysis and Debugging of Network Connectivity Policies.* Technical Report MSR-TR-2014-102. Microsoft.

[16] Siva Kesava Reddy Kakarla, Alan Tang, Ryan Beckett, Karthick Jayaraman, Todd Millstein, Yuval Tamir, and George Varghese. 2020. Finding Network Misconfigurations by Automatic Template Inference. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 999–1013. https://www.usenix.org/conference/nsdi20/presentation/kakarla

[17] Peyman Kazemian, George Varghese, and Nick McKeown. 2012. Header Space Analysis: Static Checking for Networks. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation* (San Jose, CA) *(NSDI'12)*. USENIX Association, Berkeley, CA, USA, 9–9. http://dl.acm.org/citation.cfm?id=2228298.2228311

[18] Ahmed Khurshid, Wenxuan Zhou, Matthew Caesar, and P. Brighten Godfrey. 2012. Veriflow: Verifying Network-wide Invariants in Real Time. *SIGCOMM Comput. Commun. Rev.* 42, 4 (Sept. 2012), 467–472. https://doi.org/10.1145/2377677.2377766

[19] TOM Krazit. 2019. Networking issues take down Google Cloud in parts of the U.S. and Europe, YouTube and Snapchat also affected. https://www.geekwire.com/2019/networking-issues-take-google-cloud-parts-u-s-europe-youtube-snapchat-also-affected/

[20] Nuno P. Lopes, Nikolaj Bjørner, Patrice Godefroid, Karthick Jayaraman, and George Varghese. 2015. Checking Beliefs in Dynamic Networks. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. USENIX Association, Oakland, CA, 499–512. https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/lopes

[21] Haohui Mai, Ahmed Khurshid, Rachit Agarwal, Matthew Caesar, P Brighten Godfrey, and Samuel Talmadge King. 2011. Debugging the data plane with anteater. *ACM SIGCOMM Computer Communication Review* 41, 4 (2011), 290–301.

[22] Timothy Nelson, Christopher Barratt, Daniel J. Dougherty, Kathi Fisler, and Shriram Krishnamurthi. 2010. The Margrave Tool for Firewall Analysis. In

Proceedings of the 24th International Conference on Large Installation System Administration (San Jose, CA) *(LISA'10)*. USENIX Association, USA, 1–8.

[23] Networkworld. 2015. What was wrong with United's router? https://www.networkworld.com/article/2946070/what-was-wrong-with-uniteds-router.html

[24] Santhosh Prabhu, Kuan Yen Chou, Ali Kheradmand, Brighten Godfrey, and Matthew Caesar. 2020. Plankton: Scalable network configuration verification through model checking. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 953–967. https://www.usenix.org/conference/nsdi20/presentation/prabhu

[25] Steve Ragan. 2016. BGP errors are to blame for Monday's Twitter outage, not DDoS attacks. https://www.csoonline.com/article/3138934/bgp-errors-are-to-blame-for-monday-s-twitter-outage-not-ddos-attacks.html

[26] David A. Ramos and Dawson R. Engler. 2011. Practical, Low-Effort Equivalence Verification of Real Code. In *Computer Aided Verification*, Ganesh Gopalakrishnan and Shaz Qadeer (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 669–685.

[27] STAN Schroeder. 2013. Facebook Suffers Sitewide Errors for Many Users. https://mashable.com/2013/10/21/facebook-currently-doesnt-allow-status-updates/

[28] Ruchit Shrestha, Xiaolin Sun, and Aaron Gember-Jacobson. 2020. Localizing Router Configuration Errors Using Unsatisfiable Cores. (2020).

[29] Bingchuan Tian, Xinyi Zhang, Ennan Zhai, Hongqiang Harry Liu, Qiaobo Ye, Chunsheng Wang, Xin Wu, Zhiming Ji, Yihong Sang, Ming Zhang, Da Yu, Chen Tian, Haitao Zheng, and Ben Y. Zhao. 2019. Safely and Automatically Updating In-Network ACL Configurations with Intent Language. In *Proceedings of the ACM Special Interest Group on Data Communication* (Beijing, China) *(SIGCOMM '19)*. Association for Computing Machinery, New York, NY, USA, 214–226. https://doi.org/10.1145/3341302.3342088

[30] DYLAN TWENEY. 2013. 5-minute outage costs Google $545,000 in revenue. https://venturebeat.com/2013/08/16/3-minute-outage-costs-google-545000-in-revenue/

[31] Konstantin Weitz, Doug Woos, Emina Torlak, Michael D. Ernst, Arvind Krishnamurthy, and Zachary Tatlock. 2016. Scalable Verification of Border Gateway Protocol Configurations with an SMT Solver. *SIGPLAN Not.* 51, 10 (Oct. 2016), 765–780. https://doi.org/10.1145/3022671.2984012

[32] Hongkun Yang and Simon S Lam. 2015. Real-time verification of network properties using atomic predicates. *IEEE/ACM Transactions on Networking* 24, 2 (2015), 887–900.

[33] Fangdan Ye, Da Yu, Ennan Zhai, Hongqiang Harry Liu, Bingchuan Tian, Qiaobo Ye, Chunsheng Wang, Xin Wu, Tianchen Guo, Cheng Jin, Duncheng She, Qing Ma, Biao Cheng, Hui Xu, Ming Zhang, Zhiliang Wang, and Rodrigo Fonseca. 2020. Accuracy, Scalability, Coverage: A Practical Configuration Verifier on a Global WAN. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication* (Virtual Event, USA) *(SIGCOMM '20)*. Association for Computing Machinery, New York, NY, USA, 599–614. https://doi.org/10.1145/3387514.3406217

[34] Ennan Zhai, Ang Chen, Ruzica Piskac, Mahesh Balakrishnan, Bingchuan Tian, Bo Song, and Haoliang Zhang. 2020. Check before You Change: Preventing Correlated Failures in Service Updates. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 575–589. https://www.usenix.org/conference/nsdi20/presentation/zhai

Appendices are supporting material that has not been peer-reviewed.

Appendix

Theorem 3.3 (Soundness). *If networks $\mathcal{N}$ and $\mathcal{N}^*$ are locally equivalent for isomorphism $\mathcal{I}$, then they have the same set of routing solutions.*

Proof. The proof is by a reduction to the stable routing problem [4]. First, we show that each protocol $p \in \mathcal{P}$ forms a stable routing problem (SRP). In particular for any given destination router $d \in \mathcal{V}$ advertising initial route d_r, $I(d) \in \mathcal{V}^*$ must also advertise d_r since the protocol-specific advertisement configurations must be the same. Given this, we can construct the SRP $(\mathcal{T}, \mathcal{R}, d_r, \preceq_p, \text{trans})$ for $\mathcal{N}$ and $(\mathcal{T}^*, \mathcal{R}, d_r, \preceq_p, \text{trans}^*)$ for $\mathcal{N}^*$, where:

$$\begin{array}{lcl} \text{trans}(e,r) & = & \mathcal{F}_p(C_p(e), e, r) \\ \text{trans}^*(e,r) & = & \mathcal{F}_p^*(C_p^*(e), e, r) \end{array}$$

We further relate the two SRPs with the abstraction (f, h) where $f(e) = \mathcal{I}(e)$ and $h(r) = r$.

The main theorem for abstract SRPs is that of equivalent routing solutions when the abstractions are sound [4]. Thus, we must simply prove that this is a sound abstraction. To do so, we prove each of the sufficient conditions in [4]:

Dest-equivalence. We have $f(d) = \mathcal{I}(d)$ which is the destination router for $\mathcal{N}^*$ and $f(x) \neq I(d)$ for any $x \neq d$ by virtue of $\mathcal{I}$ being an isomorphism.

Orig-equivalence. We have $h(d_r) = d_r$ since h is the identify function, which by construction is the route used at $\mathcal{N}^*$.

Drop-equivalence. We have $h(r) = r$ since h is the identity function, which trivially satisfies the drop-equivalence requirement that $h(r) = \bot \iff r = \bot$.

Rank-equivalence. By definition, we have $r_1 \preceq_p r_2 \iff h(r_1) \preceq_p h(r_2)$ since h is the identity function.

Trans-equivalence. From the fact that $\mathcal{N}$ and $\mathcal{N}^*$ are equivalent for $\mathcal{I}$, it follows that $\mathcal{F}_p(C_p(e), e, r) = \mathcal{F}_p^*(C_p^*(I(e)), I(e), r)$. This means that we have $\text{trans}(e, r) = \text{trans}^*(I(e), r)$ by definition. Substituting the definition of f and h, this gives us the equivalence: $h(\text{trans}(e, r)) = \text{trans}^*(f(e), h(r))$, which is the desired result.

Topology-abstraction. Finally, the topology requirements from [4] are trivially satisfied since $\mathcal{I}$ is a homomorphism.

This result demonstrates that each protocol will compute the same set of routing solutions. Thus the composition of the protocols will also compute and select the same set of routes.

□