CAMPION: Debugging Router Configuration Differences

Abstract

Tools for verifying router configuration files have improved dramatically but do not offer facilities for *localizing* bugs. This paper introduces two core ideas: (1) a modular approach to finding differences in configuration components (e.g., route maps or access control lists) and localizing them to lines within the configuration, and (2) an algorithm to rewrite a set of headers from a symbolic representation with binary decision diagrams to a more operator-understandable form. We implement the debugging algorithms in the tool CAMPION, and apply them to several use cases. These include debugging pairs of backup routers from different manufacturers, validating replacement of critical routers, providing concise semantic differences, and eliminating collateral damage after configuration editing. We describe results of running CAMPION on a large cloud vendor and a university network. CAMPION was used to test 30 proposed router replacements in the cloud vendor and proactively detected four configuration bugs. One such bug on a route reflector would have caused a severe outage had it not been detected. On the university network, with backup routers from Cisco and Juniper, CAMPION found multiple bugs that were validated by operators. These bugs were undetected for 3 years and depend on subtle semantic differences that the operators said were "highly unlikely" to detect by "just eyeballing the configs."

1 Introduction

Networks today are configured through low-level configuration directives at individual routers that enforce complex policies for access control and routing. It is no wonder that network operators routinely introduce subtle configuration errors that induce costly and disruptive outages [9, 20, 25, 27, 29, 32]. In response, researchers have developed network verification tools [1] that can proactively analyze a network for correctness [1, 5, 6, 13, 14, 18, 19, 23, 26, 38–40]. Existing network verification tools can effectively *detect* the presence of configuration errors. However, routers can have thousands of lines of configurations, so the feedback the tools provide is not enough, on its own, to *debug* the causes of these errors. First, verification tools typically provide only a *single*, *concrete* counterexample (e.g., a particular packet) for a policy violation, leaving to the operator the difficult task of generalizing this example to determine the scope and impact of the violation. Second, verification tools do not localize identified errors to relevant parts of the configuration, forcing operators to determine the location of the error, another difficult task.

In this paper we tackle these debugging challenges in the context of a specific, but common, verification task: checking *behavioral equivalence of two individual router configurations*. This task arises often as part of regular network operations and maintenance. First, it is common for pairs of routers to be intended to behave equivalently, in order 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. Second, small configuration updates to individual routers are frequent, serving to enable new services, implement security updates, or perform planned maintenance. Network operators must ensure that each such update does not have any unintended behavioral impact on the original configuration. The first use case is an example of the need for behavioral equivalence checking in *space*, while the second is an example of the need for such checking in *time*.

It might appear that some form of textual configuration *diffing* suffices to identify violations of intended equivalence while identifying relevant configuration snippets. However, that approach fails for both use cases above. For the first case, it is common to use backup routers from different vendors in order to prevent vendor-specific hardware and firmware bugs from bringing down both routers simultaneously. Since each vendor has its own configuration language, with its own constructs and structures, a textual diff cannot easily determine whether the backups have equivalent behavior. For the second case, when checking correctness of a configuration update, the textual diff is already known. Rather, we desire to understand the *behavioral* impact of the change versus the original configuration to prevent unintentional effects.

Our tool, called CAMPION, addresses the above need. CAMPION takes as input a pair of router configurations that are intended to be (mostly) equivalent, and it outputs a set of *behavioral differences* as well as information to aid in debugging or understanding those differences. For each difference, CAMPION identifies the set of inputs (e.g., packets) that are affected, the set of configuration lines in each configuration that affect these inputs, and the behavioral difference that occurs (e.g., accept or reject). CAMPION does this by doing a modular comparison between corresponding components in the two configurations. It currently identifies differences in the two most important and complex configuration component types: access-control lists (ACLs), which determine the data plane packets allowed in and out of each network interface, and route maps, which determine the route advertisements

[1] We contrast tools for debugging *systems* configurations from tools for verifying *router* configurations in §6.

sent in the control plane.[2]

CAMPION contains two key algorithms and uses a novel modular approach with an underlying formal basis. The first algorithm, CHANGEDETECT, takes two configuration components, C_1 and C_2 (both ACLs or both route maps), detects all behavioral differences, and localizes those differences to configuration lines. To do this, we model C_1 and C_2 as if-then-else structures and consider all paths through the two components. For each path p_1 through C_1 and each path p_2 through C_2, we check whether there is some input (a packet or route advertisement) that traverses along p_1 and p_2 through their respective components and exhibits a behavioral difference. Modeling the components' behaviors, instead of using just the text, allows for comparison of configurations from different vendors, and comparing the behavior on specific paths allows the results to be mapped back to the configuration lines along those paths. For each pair of paths, the set of inputs are represented symbolically with binary decision diagrams (BDDs) to make the pairwise path exploration efficient. Our CHANGEDETECT algorithm is conceptually similar to prior approaches to checking equivalence, for example of C functions [28] and network data planes [11]. To our knowledge ours is the first approach that can precisely check equivalence of network control-plane structures, notably route maps.

While CHANGEDETECT identifies behavioral differences and localizes them to configuration lines, our second algorithm, HEADERPRESENT, localizes differences to an operator-understandable space of inputs. Traditionally, both network verification tools and equivalence checkers simply output a single example input that illustrates each difference. Instead, HEADERPRESENT produces a concise and intuitive representation of the *full set* of affected inputs for each difference. Given a set I of inputs in the form of a BDD, HEADERPRESENT does this by representing I in terms of the constants (prefixes or prefix ranges) that appear in the configurations, and doing so in a minimal way.

CAMPION uses these two algorithms to detect and debug violations of behavioral equivalence. Given two router configurations, CAMPION applies CHANGEDETECT for each pair of corresponding route maps and ACLs in the configurations. The result is the set of all per-component path pairs that exhibit a behavioral difference. CAMPION then applies HEADERPRESENT to the set of inputs associated with each such path pair. Finally, the operator is provided a comprehensive result characterizing all behavioral differences in these components and localizing them both to the relevant configuration lines and relevant input spaces.

We evaluated CAMPION on the network configurations of a large cloud provider, as well as the network configurations of a large university campus. We highlight two key results; full details are 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 severe. Second, the university network has a pair of core routers and a pair of border routers that are from different device vendors and intended to be backups of one another. CAMPION identified and localized six confirmed configuration errors across these two pairs. Further, these errors have been present in the configurations for nearly three years, which demonstrates the difficulty of manually identifying behavioral differences. CAMPION only takes few seconds to compare a pair of routers.

To summarize, the contributions of this paper are:

- We present an approach to equivalence checking of network configuration components that localizes differences both to the relevant configuration lines and to the relevant sets of inputs. Our approach leverages two new algorithms that we call CHANGEDETECT and HEADERPRESENT.
- We prove a theorem formalizing the fact that our modular approach to equivalence checking of configuration components can be used to check end-to-end network behavioral equivalence, despite not modeling or reasoning about the network protocols. Existing control-plane verification tools are monolithic and hence make it hard to localize errors. Our modular approach enables much better localization, but the tradeoff is that it can be subject to false positives, since local behavior differences at the component level do not necessarily imply global behavior differences at the router level. However, our empirical results show that local differences are relevant and indicative of errors, as our approach matches the modular way in which configurations are written.
- We have implemented our approach in the tool CAMPION (§4), which does automated behavioral equivalence checking for router configurations, including configurations from multiple vendors.
- We present an experimental evaluation of CAMPION on routers from a large cloud vendor and a university network (§5). In one experiment, the operators of the cloud vendor ran CAMPION to proactively test 30 proposed router replacements and proactively detected four configuration bugs. One such bug on a route reflector would have caused a severe outage had it not been detected. On the university network, with backup routers from Cisco and Juniper, CAMPION found multiple bugs that were validated by operators. These bugs were undetected for 3 years and depend on subtle semantic differences that the operators said were "highly unlikely" to detect by "just eyeballing the configs."

2 CAMPION by Example

This section describes an example of CAMPION's output when checking the behavioral equivalence of configurations. It then

[2] Data plane refers to processes that forward packets. Control plane refers to processes that determine which path to use.

demonstrates the advantages of CAMPION by comparing its output to that of Minesweeper [5] – a state-of-the-art network configuration verification tool.

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. Consequently, the route maps are structurally similar. 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 [3] 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 accept all other advertisements (7-12 in Figure 1(a) and 6-21 in Figure 1(b)).

```
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

Despite the superficial similarity of the two configurations, there are differences. Table 1 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. Between the two results, 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*.

	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 1: Output produced by CAMPION

In the output shown in Table 1(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

[3] A community tag is a way to color a set of routes so that they can be treated uniformly. For more details refer to RFC 1997 [21].

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 Cisco router does not forward

Table 2: Relevant information produced by Minesweeper when checking equivalence of configurations from Figure 1

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.

The second result that CAMPION produces (Table 1(b)) identifies an additional, 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 identifies 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: the community list `COMM` in the Cisco route map matches route advertisements containing *either* the community `10:10` or `10:11`, whereas the community list `COMM` in the Juniper route map erroneously matches only advertisements tagged with *both* communities.

This example was based on route maps used in real routers, and the campus network operators confirmed both of the above behavioral differences as configuration errors. Further, the errors identified by CAMPION are quite subtle and have existed in the configurations since at least July 2017. As the network operator commented, “your config-analysis tool is great. It’s highly unlikely anyone would detect the functional discrepancies just by eyeballing the configs.” As described in §5.2, CAMPION found additional differences, but these have been removed to not overcomplicate the example.

2.1 Comparison with Control-Plane Verifiers

Users could adapt existing network *control-plane verification* approaches, such as Minesweeper [5] and Bagpipe [34] to check equivalence of two route maps. These techniques build a logical representation of the router configurations as well as the corresponding routing process that creates the forwarding rules for each router, and they then use a satisfiability modulo theories (SMT) solver to answer verification queries. However, unlike CAMPION, when verification fails they only obtain concrete counterexamples from the SMT solver and provide no reference to the original configurations. They do not attempt to perform either *header localization* or *text localization*.

To illustrate the benefits of CAMPION over these approaches, we adapted Minesweeper to check route-map equivalence by using an SMT query to check that whenever the two route maps receive the same set of route advertisements, they produce the same forwarding behavior for all packets. Table 2 shows Minesweeper’s output 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 1(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 1(a) and Table 1(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 explicit also provides an indication of the scope of the problem. (3) Minesweeper does not provide any information about what parts of the configurations are responsible for the behavioral difference. This makes debugging difficult in configurations with large numbers of lines or complex policy.

It is possible to enhance Minesweeper to produce additional concrete examples, by simply running the tool 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 examples from both classes of differences from Table 1 but it takes 7 runs of the above method before it finds at least one example 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 clear violation of Difference 1 instead of Difference 2.

3 Design and Algorithms

We now present CAMPION’s design and core algorithms. CAMPION’s overall algorithm for identifying behavioral differences between two configurations C_1 and C_2 is as follows:

The 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 MATCHPOLI-

```
1  func ConfigDiff (C1, C2)
2      result ← [ ]
3      pairs ← MatchPolicies(C1, C2)
4      for (p1, p2) ∈ pairs do
5          differences ← ChangeDetect(p1, p2)
6          for d ∈ differences do
7              result ← result.append(HeaderPresent(d, {C1, C2}))
       return result
```

CIES 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, CHANGEDETECT produces a set of *differences*, each of which includes a set of inputs, the actions taken by each component, and the relevant text lines in the configurations.
3. For each such difference, HEADERPRESENT transforms the corresponding set of inputs into a more operator-understandable form that is expressed in terms of constants from the configuration.

In the remainder of the section, we describe the two algorithms CHANGEDETECT and HEADERPRESENT in more detail. We then show that these algorithms have wider applicability than use for configuration comparison by presenting an application to single-router debugging. Finally, CAMPION currently supports ACLs and route maps but we describe how our modular, component-by-component approach can be extended to find and localize behavioral differences across entire router configurations.

3.1 ChangeDetect

CHANGEDETECT takes a pair of configuration components c_1 and c_2 as input and returns a list of all behavioral differences between the two components. The same basic algorithm applies to both ACLs and route maps. Each resulting 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 CHANGEDETECT algorithm has two main steps. First, for each configuration component, the space of inputs is divided into equivalence classes, based on their *path* 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 ⟦`NETS`⟧ to denote the set of accepted prefixes, and similarly ⟦`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 ⟦`NETS`⟧, the second clause is associated with ¬ ⟦`NETS`⟧ ∩ ⟦`COMM`⟧, the space of routes matching ⟦`COMM`⟧ but not ⟦`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.

```
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.

Once the inputs are partitioned into equivalence classes for both components, the CHANGEDETECT 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 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 CHANGEDETECT.

3.2 HeaderPresent

CHANGEDETECT produces the set of all behavioral differences between two configuration components. However, the set of inputs that exhibit each difference is represented as an arbitrary logical predicate. The HEADERPRESENT algorithm produces a more human-understandable representation of this predicate by expressing it in terms of the constants (e.g. IP prefixes) that appear in the configuration components, handling the *header localization* problem. Specifically, HEADERPRESENT determines the set of packets relevant to an ACL

difference and the set of IP prefixes relevant to a route map difference. For clarity, in this section we focus on finding prefix ranges relevant to a route map differences. However, this process only relies on the constants from the configuration being sets, so it can be easily adapted to finding a representation for the data packets relevant to an ACL difference. In principle, HEADERPRESENT can also be extended to other route fields such as communities, but we has not yet implemented this. 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 (`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 HEADERPRESENT is the logical representation of a set S of prefixes, in this case a BDD representing the inputs relevant to a policy difference, along with the original configurations C_1 and C_2. The output is a representation of S 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 HEADERPRESENT is to identify the minimal such representation.

To find this minimal representation, HEADERPRESENT 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 [10], but here we associate each node with prefix ranges rather than tri-state bit vectors representing data plane packets. HEADERPRESENT'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 the 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 [10]. 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 HEADERPRESENT 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.

```
1  func GETMATCH (S, node)
2      C ← CHILDREN(node)
3      R ← PREFIXRANGE(node)
4      if ISLEAF(node) then
5          if R ⊆ S then
6              return {R}                        ▶ node is a leaf, and R ⊆ S
7          else
8              return ∅                          ▶ node is a leaf, and R ∩ S = ∅
9      if REMAINDER(node, C) ⊆ S then    ▶ checks if R − C1 − C2 ... Ck ⊆ S
10         nonmatches ← ⋃_{k∈C} GETMATCH(¬ S, k)
11         return {R − nonmatches}               ▶ returns {R − X1 − X2 ... Xm}
12     else
13         return ⋃_{k∈C} GETMATCH(S, k)         ▶ returns {X1, X2 ... Xn}
```

The GETMATCH algorithm above 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

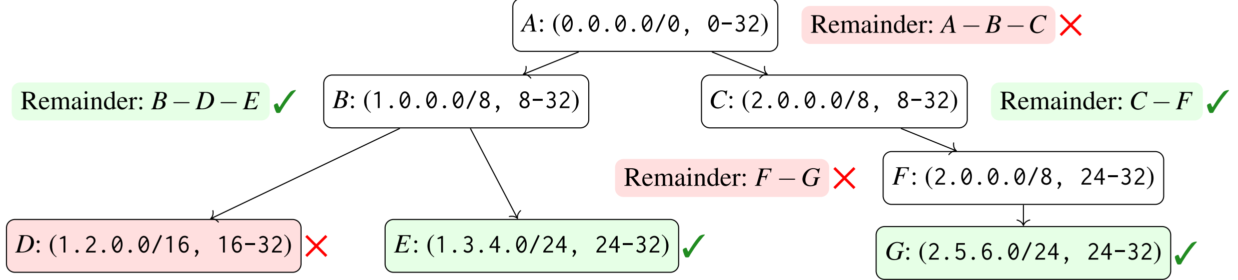

Figure 3: DAG created from prefix ranges. Green (✓) nodes represent leaves or remainders contained in a set S, and red (✗) nodes represent those that are not. S can be represented by the union of $B-D$, $C-F$, and G

process. As a final 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 $\{B-D, C-F, G\}$.

3.3 Header Project

We have shown how CHANGEDETECT and HEADERPRESENT algorithms can be used to find and localize behavioral differences. However, these algorithms have wider applicability as *debugging primitives* for network operators. We show how to use these primitives to provide a new capability that we call HEADERPROJECT, which localizes the parts of a configuration component relevant to a given space of message headers. Operators often learn that certain packets are not getting through the network, and they need to understand if the network is truly at fault, and if so then where the problem lies. HEADERPROJECT is targeted at such troubleshooting.

```
1  func HEADERPROJECT (c, H, C)
2      results ← [ ]
3      notAccepted ← CHANGEDETECT(c, acceptAll)
4      notRejected ← CHANGEDETECT(c, rejectAll)
5      allPaths ← notAccepted ∪ notRejected
6      for p ∈ allPaths do
7          results ← results.append(HEADERPRESENT(p, {C}))
8      results ← RESTRICT(results, H)
9      return results
```

Above, we show HEADERPROJECT for a component c, space H, and configuration C. It uses CHANGEDETECT to compare the given configuration component against components that simply accept or reject all messages. The union of these results therefore has the effect of breaking the input space into equivalence classes based on the paths through c. We then use HEADERPRESENT to convert each input space into an operator-understandable representation. Finally the RESTRICT function selects the subset whose input spaces have a non-trivial overlap with the space H of interest.

3.4 Debugging an Entire Router

CAMPION identifies behavioral differences between the ACLs and route maps of two router configurations. These are the configuration components that implement the major parts of

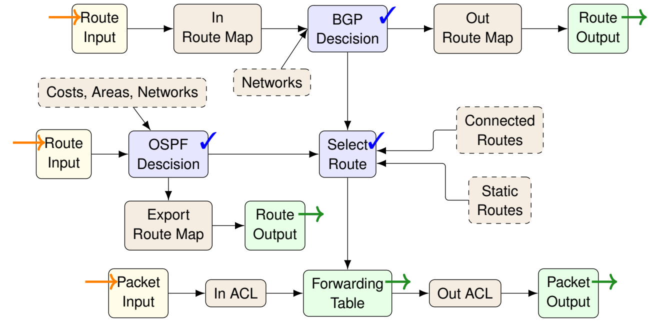

Figure 4: Basic features of routing and forwarding. Blue nodes(✓) represent fixed processes. Yellow nodes (incoming →) are inputs and green nodes (outgoing →) are outputs. Unmarked (brown) nodes represent configurable entities among which the dashed rectangle nodes are not handled by the tool.

a router's policy (and CAMPION has found and localized a rich class of errors in these components, see §5). However, CAMPION's modular, component-by-component equivalence checking can be extended to entire router configurations. Here we describe the aspects of router configurations that would need to be handled, and we also formalize the correctness of this approach to checking router equivalence.

Figure 4 provides a flow diagram illustrating the some of the routing and forwarding processes of a router. For routing, we show both a BGP process (top of the figure) and an OSPF process (middle of the figure), as these are the most common inter-domain and intra-domain routing protocols, but other protocols could be similarly added. 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 or packets(in yellow (incoming →)) or outputs and byproducts like selected routes and forwarded packets(in green (outgoing →)).

Our observation is that many crucial parts of routing and forwarding are fixed, such as the route selection process. All of the various features in Figure 4 need to be modelled in some way to fully simulate a router or network, but for finding behavioral differences, only the configured aspects with nodes

in brown need to be modelled. These can be compared in a modular fashion. The solid brown nodes represent parts of the configuration that CAMPION handles, while the dashed brown nodes are parts that are currently not supported. The latter include Connected Routes, Static Routes, OSPF Costs and networks advertised. However, unlike route maps and ACLs, these parts of a network configuration have an if-then-else structure. Some, like static routes, can be thought of as sets, with two configurations being equivalent if they both define the same number of static routes to the same destinations with identical parameters. Others can be single values, which are even simpler. Hence, localizing differences in these parts of a router configuration can be done without any complicated modelling. We plan to add this in a future version.

Once CAMPION is extended in this way, it will be a sound verifier for router configuration equivalence: If CAMPION identifies no differences, then the two router configurations are behaviorally equivalent. We have formalized CAMPION's modular approach to checking router equivalence:

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* $C_{\mathcal{P}} : \mathcal{E} \to \Omega$ *that maps each edge in the topology to a configuration* Ω*, a family of transfer functions* $\mathcal{F}_{\mathcal{P}} : \Omega \times \mathcal{E} \times \mathcal{R} \to \mathcal{R}$ *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. *Given 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* 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* $\mathcal{F}_p(C_p(e), e, r) = \mathcal{F}^*_p(C^*_p(I(e)), I(e), r)$*.*

Theorem 3.1 (Soundness)**.** *If networks* $\mathcal{N}$ *and* $\mathcal{N}^*$ *are locally equivalent for isomorphism* I*, then they have the same set of routing solutions.*

Proof. The proof is by a reduction to the stable routing problem [6]. Because an isomorphism is a homomorphism, $\mathcal{N}^*$ can be viewed as an abstraction of $\mathcal{N}$ with I serving as the topology abstraction function and the with identity abstraction function for routes $\mathcal{R}$. The full proof is in the appendix. □

4 Implementation and Limitations

Our implementation uses the Batfish [13] network configuration tool to parse configuration from different vendors into a vendor-independent representation. CAMPION operates on this representation, allowing configurations from different vendors to be compared. Sets of packets and route advertisements are represented by BDDs. These are handled with the JavaBDD library, extending code from Bonsai [6] used to encode import filters, output filters, and ACLs.

CAMPION can identify differences and perform header localization for any vendor format that Batfish supports. However, currently CAMPION can only output relevant text lines for configurations in the Cisco IOS and Juniper JunOS formats, since we must write *unparsers* to convert Batfish's representation back to configuration text. Some of the test cases in the evaluation section below use Cisco IOS XR routers, which have their own configuration format. For these cases, CAMPION does not produce text lines, but it still provides substantial localization information, including the affected headers and the resulting actions.

HEADERPRESENT for route maps currently only provides exhaustive information for IP prefix ranges. For other relevant parts of a route advertisement CAMPION provides a single example. For example, the second difference in Table 1 provides an example community tag `10:10`. CAMPION also does not support route maps with AS-path filtering. On the other hand, HEADERPRESENT for ACLs identifies the relevant prefixes, destination and source port numbers, and protocols.

5 Evaluation

We applied CAMPION to router configurations from a large cloud provider and the campus network of a large university. Cloud network A deliberately employs a diversity of hardware vendors, including within pairs of redundant routers. Our experiments on network A demonstrate CAMPION's ability to identify cross-vendor configuration differences in cloud data centers and to provide actionable localization information to operators. The university network also employs backup pairs from different vendors, but within a campus network architecture rather than a data center. We had direct access to the campus configurations and used these to dissect the found bugs in detail. Finally we illustrate how CAMPION can be used to debug configuration generation systems by applying it to Google's Capirca tool.

5.1 Data Center

Network A is from a global cloud vendor that uses routers from different manufacturers. The data center network we tested CAMPION on from vendor A employs a Clos topology consisting of hundreds of routers and thousands of servers. All the routers in this data center network are from either Juniper or Cisco, for which the configuration languages are supported by CAMPION. The data center network uses a combination of eBGP, iBGP, IS-IS, static routes, ACLs, and route redistribution for layer-3 routing topology, and it carries business traffic of multiple global services. Each router has a configuration with thousands of lines.[4]

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 BGP policies. For diversity, these redun-

[4]We do not reveal the precise number for confidentiality reasons.

dant routers are provided by different vendors such as Juniper and Cisco. It is important to not only ensure the equivalence of multi-vendor, redundant routers, but also to *quickly* localize root causes resulting in inequality. We used CAMPION to compare route maps of all redundant router pairs in a datacenter network with tens of pairs of backup routers.

Scenario 2: Router replacement. Provider *A* has an important type of network operation called *router replacement*. In a router replacement operation, the 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/features of newer routers. For example, the operators of provider *A* might replace lower-version Cisco routers with higher-version Juniper routers in order to avoid a bug in the Cisco routers. Router replacement is one of the most risky update operation tasks in provider *A*, since operators must manually rewrite the old configurations to the new format, and many critical errors have occurred as a result. The operators used CAMPION to check for difference between old and new configurations before performing a scheduled replacement, in order to *proactively* detect errors.

Scenario 3: Access control in gateway routers. The previous two scenarios leverage CAMPION's ability to analyze route maps. In network *A*, many ACL rules are applied in gateway routers for traffic control. All of provider A's gateway routers should have identical access-control policies. It is difficult for provider A's operators to guarantee the equality of ACL rules across multi-vendor gateway routers 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 equality of ACL rules in the gateway routers of our selected datacenter.

Output evaluation. Note that Provider *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.

Scenario 1: Debugging redundant routers. CAMPION detected five configuration bugs across all of the redundant router pairs that it analyzed. All of the bugs represent missing fragments of BGP policy in one router. For four of them, CAMPION was able to accurately localize the difference. For example, CAMPION pointed out that one of the prefixes for the import filter was missing in the primary router but present in the backup one. An immediate question is why these bugs did not result in routing problems that would have been detected by customers or other real-time monitoring systems? This was because the missing prefixes had not been used for production traffic yet, but would be in the near future. Once the traffic of some service using this prefix would be enabled, a service problem would have occurred. Thus, CAMPION prevented a future service disruption.

The fifth error that CAMPION detected used a variation of Cisco IOS that CAMPION does not properly unparse from Batfish's vendor-independent format. As a result CAMPION produced useful localization information, such as the relevant input space and the actions taken by each router, along with some inaccurate configuration text. Due to this inaccuracy, the operator reported the need to spend more time to understand the precise bug location, but still said that it was easy to spot the deviant configuration lines from CAMPION's output.

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 of the local preference bugs 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.

Scenario 3: Access control in gateway routers. CAMPION successfully detected three ACL differences between gateway routers from Cisco and Juniper. Table 3 shows CAMPION's output for one of these differences.[5] CAMPION identified the exact line in the Cisco ACL where traffic is rejected. The Juniper equivalent of ACLs is divided into terms, and CAMPION is able to locate which term accepted the traffic. It also provided exact header information, identifying the relevant source IP prefix. The Excluded Packets row identifies some packets that contained in the relevant source IP prefix but were not matched by the provided text lines because they were matched by earlier lines in the ACLs.

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

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 the checking task within 5 seconds per pair of routers.

Comparing CAMPION with an existing tool. As a global cloud service provider, provider *A* has deployed its own homegrown verification system for checking routing behavior correctness for its global network. This system has been used

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

for 1.5 years. However, while this verification system can tell whether the network configuration meets the operators' intent, it does not provide any debugging or error localization capability. Thus, provider *A*'s operators spend considerable time localizing bugs even when the tool identifies bugs in the network. CAMPION therefore provides a new capability that can potentially reduce debugging time considerably for provider *A*'s operators.

Localization efficiency. For the configuration pairs checked by operators, all the localization results are less than five lines of configuration code. Each of the configuration files tested vary in size ranging from about 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 when attempting to fix or understand a difference.

5.2 University Network

The university network consists of approximately 1400 devices, including border routers that connect to external ISPs, core routers that form the backbone, and building routers that handle connectivity for individual buildings.

We ran CAMPION to compare the BGP routing policies for a pair of core routers and a pair of border routers. In each pair, one of the configurations uses a Cisco configuration format, and the other uses the Juniper format. These two pairs were chosen because they are the only Cisco-Juniper backup pairs with BGP policy in the network. 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. According to the network operators, the two routers are intended to have the same behavior for each corresponding neighbor, but the names of the route maps differ between the two routers. To determine which route maps correspond, we match the ones 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. We found differences in all of the export route maps and notified the network operators. The results are summarized in Table 4.

The operators did not directly use CAMPION. Instead, we looked at the results from CAMPION, briefly inspected the configurations, and produced a list of actionable items to send to the operators. We did this only because it was difficult to get time from the campus operators to learn the user interface. When inspecting the configurations, we manually examined the definitions of the affected route map to try to find any discrepancy between corresponding parts in the two configurations. The prefix ranges, communities, and text lines produced by CAMPION made it straightforward to identify these discrepancies. As a result of this process, 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 4, 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. The remaining two columns show how operators responded to the items that we reported.

Router Pair	Route Map	Outputted Differences	Differences Reported	Confirmed	Pending
Core Routers	Export 1	5	5	2	2
	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	-	-	-

Table 4: University Network

As shown in the table, the operators confirmed that most of the differences CAMPION identified were in fact errors. Based on the snapshots, it appears that 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 4). 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."

The route maps shown in Figure 1 were simplified. 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. 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. The operators are still investigating these issues. 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 were similar in that they affected the matched prefixes and communities, but were of a different nature: there were differences in two regular expressions used to match

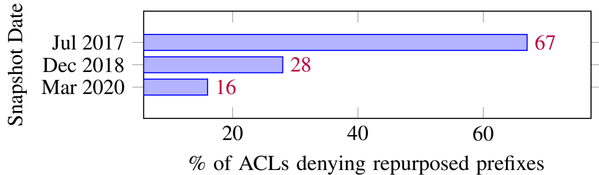

Figure 5: Use of HEADERPROJECT to identify ACLs affecting repurposed prefixes

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 corresponding Cisco router list. These were also confirmed as errors by the operators.

In a separate experiment, we evaluated HEADERPROJECT (§3.3). Prior to 2017, many ACLs in the university network were configured to reject all packets with destinations matching some infrastructure prefixes, say P. These prefixes were subsequently reallocated and those ACL lines were no longer necessary. We ran HEADERPROJECT using a regular expression to match the ACLs of interest in over 90 Cisco router configurations, and limited the results to show the behavior for P. We ran this on three different snapshots of the network; the percentage of ACLs with lines denying P are shown in Figure 5. The results display the text used to match packets, so we can count how many instances of the line "`deny ip any P`" appeared. In the earliest snapshot, we found that 67% of the ACLs still had lines explicitly rejecting packets to P. In the later snapshots, we found that 28% and 16% ACLs still had those lines, confirming that changes were made but that many ACLs have still not been properly updated.

HEADERPROJECT has many advantages. In both executions, we compared many routers across the network without manually sifting through individual configurations. The results show the exact lines that match a given packet, so we could easily determine if packets were rejected because they were part of the repurposed networks or for another reason. For configurations where the changes were successfully made, the tool showed what new traffic was now accepted, confirming that the change affected network policy. A purely text-based comparison would be unable to determine this.

5.3 ACL Compilers and Microbenchmarks

We ran CAMPION on ACLs generated from Capirca [6], a multi-platform ACL generation system created by Google. Capirca has its own language for specifying access-control policies and then generates ACLs for various different platforms. In this experiment, we generated several policy definitions that filter randomly chosen destination prefixes and port numbers. Using Capirca, we generated one version of the policy for Cisco and one for Juniper. We then compared these outputs using CAMPION. Unsurprisingly, we did not discover any difference in behavior between the two ACLs in any of the trials, as Capirca is relatively mature. However, this experiment shows the potential application of CAMPION to validate and debug configuration generation systems.

Number of ACL rules generated	Parsing time (s)	Compare time(s)	Total time(s)
10	1.18	0.37	1.55
100	1.76	0.62	2.38
1000	3.27	0.58	3.85
10000	19.37	15.03	34.37

Table 5: Processing Times for randomly generated ACLs

We also used this experiment to measure CAMPION's performance on random ACLs of differing sizes. We generated a single random ACL of a given size, made a version containing 10 edits (or 2 edits when the number of rules is 10), and ran CAMPION to compare the two ACLs. The results are shown in Table 5. Parsing time is the time for Batfish to create a vendor-independent model, while Compare time is the time for CAMPION to construct BDDs and compare ACLs. CAMPION takes only seconds for even large ACLs with thousands of lines; in all cases, the comparison time is less than the time it takes to parse the configurations. Recall the basic pairwise algorithm (CONFIGDIFF) is $O(N^2)$ where N is module size; this will not affect usability if modules are thousands of lines or less as they are even in large clouds (§5).

6 Related Work

There has been significant work recently in various aspects of network verification. At a high level our work differs from this prior work 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: Several tools verify reachability properties of a network's data plane, including its ACLs and forwarding tables [2, 15, 18, 19, 22, 23, 38]. Several tools focus on ACLs [24, 31] and localize errors to ACL lines [15, 16, 18, 31]. Closest to our work, `netdiff` [11] 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 (e.g., routing errors). Our HEADERPRESENT approach to localizing the space of inputs affected by a behavioral difference also has no analogue in `netdiff`.

Control Plane Verification Tools: Other tools verify properties of a network's control plane, which includes its routing

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

processes [1, 5, 6, 12, 13, 26, 34]. These tools can be adapted to perform equivalence checking for route maps and ACLs, as we showed for Minesweeper [5] 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, extending Minesweeper by negating already found counterexamples, may never find all bugs and still leaves the question as to which parts of the text caused the bug. CAMPION leverages the BDD encoding of ACLs and route maps from Bonsai [6], which uses BDDs to perform network abstraction not router differencing or debugging. Finally, recent work extends Minesweeper to localize errors by leveraging an SMT solver's ability to provide *unsatisfiable cores* when verification fails [30]. The approach localizes errors to specific SMT constraints, within the overall logical formula, but not to either configuration lines or headers.

Outlier Detection: Some work automatically identifies network *outliers*. Benson *et al.* [7, 8] 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 configurations. SelfStarter [17] 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 since it does not identify behavioral differences.

Debugging system configurations: A rich body of work [3, 4, 33, 35] seeks to debug *system* configuration errors reactively or proactively [37]. Reactive configuration debugging localizes the root causes of configuration errors *after* a system outage. Approaches include tracking dependencies (e.g., ConfAid [4]), binary search for errors (e.g., Chronus [35]), and signature-based analysis (e.g., PeerPressure [33]). Proactive configuration debugging tools identify potential errors *before* system deployment. For example, PCheck [36] emulates the system to detect latent configuration errors ahead of time. CAMPION focuses instead on latent configuration bugs in router configuration, whose structured segments like ACLs and route maps are arguably more complex than the typical key-value parameter settings in system configuration. However, systems code is more complex than the network forwarding and routing processes, so system configuration debugging must rely on techniques such as emulation while network configurations are amenable to automated formal verification.

Equivalence Checking: Equivalence checking is an old idea beyond networks, and our CHANGEDETECT algorithm is similar in spirit to prior work. For example, Ramos *et al.* [28] 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

We presented CAMPION, a tool for checking behavioral equivalence of network router configurations. In contrast to prior work, CAMPION localizes identified errors to the affected message headers and relevant lines of configuration text. Our experience applying CAMPION to a large cloud provider and a large university network indicates that CAMPION satisfies a real need by identifying and localizing important errors.

Prior control-plane verification tools model the entire configuration monolithically as a set of constraints. Our approach, in contrast, exploits the modular structure inherent in router configurations, where different configuration components perform different tasks (e.g., access control and routing). This "bottom up" style eases the task of localization, and sidesteps the need to reason about the routing protocols themselves. At the same time, this approach can be extended to completely cover an entire router, as formalized in Theorem 3.1, and there is no reason why it cannot in principle extend to an entire network. As in other forms of verification, we believe that exploiting modularity will be critical to making real-world network verification and debugging practical and effective.

References

[1] Anubhavnidhi Abhashkumar, Aaron Gember-Jacobson, and Aditya Akella. Tiramisu: Fast multilayer network verification. In *17th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 20)*, pages 201–219, 2020.

[2] Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. Netkat: Semantic foundations for networks. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, page 113–126, New York, NY, USA, 2014. Association for Computing Machinery.

[3] Mona Attariyan and Jason Flinn. Using causality to diagnose configuration bugs. In *2008 USENIX Annual Technical Conference, Boston, MA, USA, June 22-27, 2008. Proceedings*, pages 281–286, 2008.

[4] Mona Attariyan and Jason Flinn. Automating configuration troubleshooting with dynamic information flow analysis. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, page 237–250, USA, 2010. USENIX Association.

[5] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. A general approach to network configuration verification. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*,

SIGCOMM '17, page 155–168, New York, NY, USA, 2017. Association for Computing Machinery.

[6] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. Control plane compression. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, page 476–489, New York, NY, USA, 2018. Association for Computing Machinery.

[7] Theophilus Benson, Aditya Akella, and David Maltz. Unraveling the complexity of network management. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'09, pages 335–348, Berkeley, CA, USA, 2009. USENIX Association.

[8] Theophilus Benson, Aditya Akella, and David A. Maltz. Mining policies from enterprise network configuration. In *Proceedings of the 9th ACM SIGCOMM Conference on Internet Measurement*, IMC '09, pages 136–142, New York, NY, USA, 2009. ACM.

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

[10] Nikolaj Bjørner, Garvit Juniwal, Ratul Mahajan, Sanjit A. Seshia, and George Varghese. ddnf: An efficient data structure for header spaces. In *Haifa Verification Conference (HVC), 2016*, August 2016.

[11] Dragos Dumitrescu, Radu Stoenescu, Matei Popovici, Lorina Negreanu, and Costin Raiciu. Dataplane equivalence and its applications. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 683–698, Boston, MA, February 2019. USENIX Association.

[12] Seyed K. Fayaz, Tushar Sharma, Ari Fogel, Ratul Mahajan, Todd Millstein, Vyas Sekar, and George Varghese. Efficient network reachability analysis using a succinct control plane representation. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[13] Ari Fogel, Stanley Fung, Luis Pedrosa, Meg Walraed-Sullivan, Ramesh Govindan, Ratul Mahajan, and Todd Millstein. A general approach to network configuration analysis. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 469–483, Oakland, CA, May 2015. USENIX Association.

[14] Aaron Gember-Jacobson, Raajay Viswanathan, Aditya Akella, and Ratul Mahajan. Fast control plane analysis using an abstract representation. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, pages 300–313, New York, NY, USA, 2016. ACM.

[15] Karthick Jayaraman, Nikolaj Bjørner, Geoff Outhred, and Charlie Kaufman. Automated analysis and debugging of network connectivity policies. *Microsoft Research*, pages 1–11, 2014.

[16] Karthick Jayaraman, Nikolaj Bjørner, Jitu Padhye, Amar Agrawal, Ashish Bhargava, Paul-Andre C Bissonnette, Shane Foster, Andrew Helwer, Mark Kasten, Ivan Lee, et al. Validating datacenters at scale. In *Proceedings of the ACM Special Interest Group on Data Communication*, pages 200–213. 2019.

[17] Siva Kesava Reddy Kakarla, Alan Tang, Ryan Beckett, Karthick Jayaraman, Todd Millstein, Yuval Tamir, and George Varghese. Finding network misconfigurations by automatic template inference. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 999–1013, Santa Clara, CA, February 2020. USENIX Association.

[18] Peyman Kazemian, George Varghese, and Nick McKeown. Header space analysis: Static checking for networks. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 9–9, Berkeley, CA, USA, 2012. USENIX Association.

[19] Ahmed Khurshid, Wenxuan Zhou, Matthew Caesar, and P. Brighten Godfrey. Veriflow: Verifying network-wide invariants in real time. *SIGCOMM Comput. Commun. Rev.*, 42(4):467–472, September 2012.

[20] TOM Krazit. 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/, 2019.

[21] Tony Li, Ravi Chandra, and Paul S. Traina. BGP Communities Attribute. RFC 1997, August 1996.

[22] Nuno P. Lopes, Nikolaj Bjørner, Patrice Godefroid, Karthick Jayaraman, and George Varghese. Checking beliefs in dynamic networks. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 499–512, Oakland, CA, May 2015. USENIX Association.

[23] Haohui Mai, Ahmed Khurshid, Rachit Agarwal, Matthew Caesar, P Brighten Godfrey, and Samuel Talmadge King. Debugging the data plane with anteater.

ACM SIGCOMM Computer Communication Review, 41(4):290–301, 2011.

[24] Timothy Nelson, Christopher Barratt, Daniel J Dougherty, Kathi Fisler, and Shriram Krishnamurthi. The margrave tool for firewall analysis. In *LISA*, pages 1–18, 2010.

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

[26] Santhosh Prabhu, Kuan Yen Chou, Ali Kheradmand, Brighten Godfrey, and Matthew Caesar. Plankton: Scalable network configuration verification through model checking. In *17th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 20)*, pages 953–967, 2020.

[27] Steve Ragan. 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, 2016.

[28] David A. Ramos and Dawson R. Engler. Practical, low-effort equivalence verification of real code. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, volume 6806 of *Lecture Notes in Computer Science*, pages 669–685. Springer, 2011.

[29] STAN Schroeder. Facebook suffers sitewide errors for many users. https://mashable.com/2013/10/21/facebook-currently-doesnt-allow-status-updates/, 2013.

[30] Ruchit Shrestha, Xiaolin Sun, and Aaron Gember-Jacobson. Localizing router configuration errors using unsatisfiable cores. In *Poster Session at 17th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 20)*.

[31] 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. Safely and automatically updating in-network ACL configurations with intent language. In *ACM SIGCOMM (SIGCOMM)*, 2019.

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

[33] Helen J Wang, John C Platt, Yu Chen, Ruyun Zhang, and Yi-Min Wang. Automatic misconfiguration troubleshooting with peerpressure. In *OSDI*, volume 4, pages 245–257, 2004.

[34] Konstantin Weitz, Doug Woos, Emina Torlak, Michael D. Ernst, Arvind Krishnamurthy, and Zachary Tatlock. Scalable verification of border gateway protocol configurations with an SMT solver. In Eelco Visser and Yannis Smaragdakis, editors, *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016*, pages 765–780. ACM, 2016.

[35] Andrew Whitaker, Richard S. Cox, and Steven D. Gribble. Configuration debugging as search: Finding the needle in the haystack. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design and Implementation - Volume 6*, OSDI'04, page 6, USA, 2004. USENIX Association.

[36] Tianyin Xu, Xinxin Jin, Peng Huang, Yuanyuan Zhou, Shan Lu, Long Jin, and Shankar Pasupathy. Early detection of configuration errors to reduce failure damage. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 619–634, Savannah, GA, November 2016. USENIX Association.

[37] Tianyin Xu and Yuanyuan Zhou. Systems approaches to tackling configuration errors: A survey. *ACM Comput. Surv.*, 47(4), July 2015.

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

[39] 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. Accuracy, scalability, coverage: A practical configuration verifier on a global WAN. In *SIGCOMM '20: Proceedings of the 2020 Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication, Virtual Event, USA, August 10-14, 2020*, pages 599–614, 2020.

[40] Ennan Zhai, Ang Chen, Ruzica Piskac, Mahesh Balakrishnan, Bingchuan Tian, Bo Song, and Haoliang Zhang. Check before you change: Preventing correlated failures in service updates. In *17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020,*

Santa Clara, CA, USA, February 25-27, 2020, pages 575–589, 2020.

Appendix

Theorem 3.1 (Soundness). *If networks $\mathcal{N}$ and $\mathcal{N}^*$ are locally equivalent for isomorphism I, then they have the same set of routing solutions.*

Proof. The proof is by a reduction to the stable routing problem [6]. 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:

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

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

The main theorem for abstract SRPs is that of equivalent routing solutions when the abstractions are sound [6]. Thus, we must simply prove that this is a sound abstraction. To do so, we prove each of the sufficient conditions in [6]:

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

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 I, it follows that $\mathcal{F}_p(\mathcal{C}_p(e), e, r) = \mathcal{F}^*_p(\mathcal{C}^*_p(I(e)), I(e), r)$. 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 [6] are trivially satisfied since 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.

□