Finding Network Misconfigurations by Automatic Template Inference

Siva Kesava Reddy Kakarla and Alan Tang, *UCLA;*
Ryan Beckett, *Microsoft Research;* Karthick Jayaraman, *Microsoft Azure;*
Todd Millstein, *UCLA / Intentionet;* Yuval Tamir and George Varghese, *UCLA*

https://www.usenix.org/conference/nsdi20/presentation/kakarla

This paper is included in the Proceedings of the
17th USENIX Symposium on Networked Systems Design
and Implementation (NSDI '20)

February 25–27, 2020 • Santa Clara, CA, USA

978-1-939133-13-7

Finding Network Misconfigurations by Automatic Template Inference

Siva Kesava Reddy Kakarla[1], Alan Tang[1], Ryan Beckett[2], Karthick Jayaraman[3], Todd Millstein[1,4], Yuval Tamir[1], and George Varghese[1]

[1]University of California, Los Angeles, USA. [2]Microsoft Research, Redmond, USA.
[3]Microsoft Azure, Microsoft, Redmond, USA. [4]Intentionet, Seattle, USA.

Abstract

Network verification to detect router configuration errors typically requires an explicit correctness *specification*. Unfortunately, specifications often either do not exist, are incomplete, or are written informally in English. We describe an approach to infer likely network configuration errors without a specification through a form of automated *outlier detection*. Unlike prior techniques, our approach can identify outliers even for complex, structured configuration elements that have a variety of intentional differences across nodes, like access-control lists, prefix lists, and route policies.

Given a collection of configuration elements, our algorithm automatically infers a set of parameterized *templates*, modeling the (likely) intentional differences as variations within a template while modeling the (likely) erroneous differences as variations across templates. We have implemented our algorithm, which we call *structured generalization*, in a tool called SELFSTARTER and used it to automatically identify configuration outliers in a collection of datacenter networks from a large cloud provider, the wide-area network from the same cloud provider, and the campus network of a large university. SELFSTARTER found misconfigurations in all three networks, including 43 previously unknown bugs, and is in the process of adoption in the configuration management system of a major cloud provider.

1 Introduction

Router configuration errors are a major cause of network outages [1, 17, 19, 30, 35, 37]. Accordingly, researchers have developed a variety of techniques to automatically identify such errors and/or to prove their absence. Techniques that began in academic research [16, 23, 24, 29] have migrated to industry via cloud vendors [2, 20], router vendors [3], and startups [4]. However, these approaches have an important practical limitation: users must provide an explicit, formal *specification* of the network's intended behaviors (e.g., reachability requirements) [14, 18, 23, 29]. In practice, such specifications often do not exist, and when they do exist, they tend to be informal, incomplete, and ambiguous. A few tools do not require a specification [15, 16] but then are limited to identifying *generic* configuration errors (e.g., forwarding loops, duplicate IP addresses), independent of the network's particular policy intent.

In this paper, we develop an approach to identify *network-specific* misconfigurations without a specification, through a form of outlier detection. The *bugs as outliers* paradigm [13] is natural for network configurations, since, by design, many nodes' configurations are intended to be highly similar to one another (e.g., all nodes playing the same *role* in the network). We refer to any logical unit of a configuration, such as a BGP session configuration or access-control list, as a *segment*. Given a set of configuration segments that are intended to be similar, our goal is to automatically identify likely misconfigurations and provide actionable feedback to fix them.

Prior work in outlier detection (§ 8) for network configurations [12, 28] assumes that configuration segments are intended to be *exactly equivalent* to one another. Such approaches can identify outliers in the simpler aspects of a configuration, such as the set of DNS servers and the MTU values on interfaces. However, exact equivalence is much too strong an assumption for detecting useful outliers for complex configuration segments like access-control lists and route policies. Such segments often have a variety of intentional policy differences across nodes (e.g., the treatment of local hosts or services). Hence, outlier detection must be able to distinguish intentional differences from ones that represent likely errors.

We describe a general approach to outlier detection that addresses this limitation. Given a set of configuration segments, our algorithm, which we call *structured generalization*, infers one or more segment *templates*. Each template is a segment definition that is optionally parameterized by various pieces of data (e.g., the IP address used on a particular line). These templates serve as a compact summary of the differences across network policies and induce an equivalence relation on the nodes: two nodes are considered "equivalent" if they are instantiations of the same template. In other words, we model the (likely) intentional differences between nodes as variations *within a template* while modeling the (likely) erroneous differences as variations *across templates*.

The key challenge is to infer templates that result in useful equivalence relations. This requires a delicate balance between templates that cover too few or too many configurations. Parameterization is necessary to account for intentional differences between segments. However, supporting arbitrary parameterization would lead to overly-general templates that mask important differences. Similarly, the ordering of some configuration lines (e.g., consecutive `permit` lines in an ACL) is semantically transparent. Hence, a single template should admit such reordering. However, a single template that admits arbitrary reordering (e.g., arbitrary ACL `permit` and `deny` lines) may result in masking semantic differences, thus potentially hiding critical configuration errors.

Our structured generalization meets this challenge through a novel two-level approach to matching segments with one another. *Sequence alignment* is used to align *blocks* of segment lines with one another, thereby admitting insertions and deletions but ensuring that the order of blocks is respected. However, the similarity measure for this alignment employs *bipartite matching* to match the lines within the blocks, thereby admitting line reordering and also naturally inducing template parameters wherever two matched lines differ. Our algorithm is parameterized by the definition of blocks and the cost function for bipartite matching, which respectively control the amount of reordering and parameterization. We also provide instantiations of this generic algorithm for access-control lists, prefix lists, and route policies.

We have implemented structured generalization in a tool called SELFSTARTER. We applied SELFSTARTER to identify configuration outliers in three different kinds of networks: a large number of datacenters in a cloud provider network, totaling on the order of $O(10000)$ routers; the wide area network (WAN) of a large cloud provider, containing hundreds of routers; and the department routers of a large university campus network, containing $\sim$100 routers. In these networks, we applied SELFSTARTER to three heavily-used segment types: *access-control lists* (ACLs), which contain a sequence of `permit` and `deny` lines that determine which packets to accept; *prefix lists*, which have a similar structure as ACLs and are used to determine which route announcements to accept during routing; and *route policies*, which flexibly match sets of route announcements and update them in various ways (e.g., to add a community tag).

For the datacenter, SELFSTARTER identified 1168 outliers, of which 630 were investigated and all determined to be true positives. For the wide area, SELFSTARTER identified 56 route policy outliers, of which 33 were investigated and all were determined to be true positives. As SELFSTARTER found new bugs that were previously unknown to operators, it is in the process of being adopted in the configuration management process of the WAN. However, SELFSTARTER was much less successful in identifying true positives for prefix lists in the wide area; the reasons are explained in §6. For the university network, SELFSTARTER identified 6 ACL outliers, of which 3 were investigated and all were determined to be true positives. SELFSTARTER's metatemplates made it easy for the network operators to quickly classify outliers as true/-false positives and to remediate the actual misconfigurations. Further, the templates that SELFSTARTER generated closely matched any existing "golden" templates or configurations for these networks.

We make the following contributions:

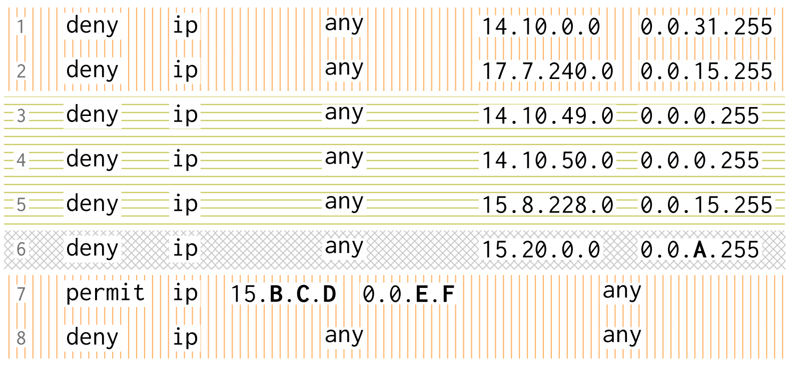
1. **Automatic Template Inference:** To our knowledge, we are the first to propose the idea of automatic template inference for network configuration segments and to employ it to identify network misconfigurations (§2).
2. **Structured Generalization:** We present a novel algorithm for automatic inference of parameterized templates for network configuration segments that combines sequence alignment and bipartite matching in a two-level structure to support controlled forms of parameterization and reordering (§3 and §4).
3. **Implementation:** We have implemented structured generalization in a practical tool called SELFSTARTER (§5).
4. **Evaluation:** We describe our empirical evaluation of SELFSTARTER on several large real-world production networks, demonstrating that in most cases it generates high-quality outliers with a low false positive rate (§6).

2 Using SELFSTARTER

We describe an example of SELFSTARTER's output and its use in finding misconfigurations in the campus network of a large university. SELFSTARTER was given as input the configurations of 106 *building routers* (one of the roles in the network), along with a regular expression capturing the name(s) of an access-control list (ACL) of interest used on the routers.[1] Given this input, SELFSTARTER automatically inferred three templates, each of which is an ACL definition that is parameterized by zero or more parameters.

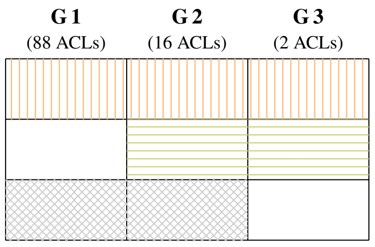
Figure 1 shows SELFSTARTER's output for this example. Figure 1(a) shows a *metatemplate*, which is a concise representation of the three inferred templates, highlighting their commonalities and differences. A metatemplate is a sequence of parameterized configuration lines. Capital letters like `A` and `B` are *parameters*, representing values that differ across ACLs that contain the corresponding line. The metatemplate is *complete*: every line that appears in some ACL is represented by a line in the metatemplate.

[1]We allow regular expressions instead of simple strings since some organizations append metadata to ACL names, so two ACLs with slightly different names may still be intended to be similar.



```
1  deny    ip                   any            14.10.0.0     0.0.31.255
2  deny    ip                   any            17.7.240.0    0.0.15.255
3  deny    ip                   any            14.10.49.0    0.0.0.255
4  deny    ip                   any            14.10.50.0    0.0.0.255
5  deny    ip                   any            15.8.228.0    0.0.15.255
6  deny    ip                   any            15.20.0.0     0.0.A.255
7  permit  ip  15.B.C.D  0.0.E.F                         any
8  deny    ip                   any                      any
```

(a) ACL Metatemplate



(b) Groups

Figure 1: SELFSTARTER output for an ACL Regex in a university network. Groups 2 and 3 are confirmed to be anomalous.

Figure 1(b) identifies the three templates that SELFSTARTER inferred. Each column represents a *group* of nodes that share a common template. Colors allow users to easily see which lines of the metatemplate belong to each template. For example, the template for the 88 ACLs in Group 1 contains the metatemplate lines that are colored orange (vertical bars) and gray (crosshatching), i.e., lines 1, 2, 6, 7, and 8. Similarly, the template for Group 2 consists of lines 1-8, and the template for Group 3 consists of lines 1-5, 7, and 8.

Though each of the example ACLs contains at most eight lines, manually scanning the configurations from all 106 routers to find outliers would be onerous and error-prone. Furthermore, partitioning the ACLs based on *exact equivalence* would result in 53 different groups, since each building has two routers with an identical ACL, but parameter values differ from building to building. In contrast, SELFSTARTER's output makes it easy for network engineers to identify outliers and to understand exactly how they differ from non-outliers. Specifically, SELFSTARTER helps engineers to identify two types of outliers, which we now describe.

Group Outliers. If SELFSTARTER produces multiple groups, the network engineer can compare the templates for these groups to decide whether one or more groups are misconfigured. Groups that are relatively small in size are particularly likely to be the results of misconfiguration. For example, in Figure 1(b) the vast majority of the ACLs are in Group 1, indicating that Groups 2 and 3 are suspect.

In fact, for this example, the network engineers have confirmed that Groups 2 and 3 represent misconfigurations. Group 3 erroneously omits line 6, allowing some flows that should be denied. Further, both Groups 2 and 3 erroneously include lines 3-5. While these lines used to be required to prevent access to certain infrastructure servers, those servers were phased out and the lines were supposed to be removed. Further, some of the denied IP addresses had since been reassigned to servers that are intended to be accessible. Hence these lines deny some flows that should be allowed.

Parameter Outliers. Structured configuration segments, like ACLs, often differ across nodes. Hence, the ability to parameterize is critical for generating templates that identify similarities without requiring exact equivalence. For each parameter, SELFSTARTER maintains a mapping from nodes to parameter values for the engineer to inspect.

A parameter error is present if some field in a line is supposed to be constant across all the routers, but it is not. SELFSTARTER guarantees that a parameter in some field of a line in the metatemplate indicates that there must be at least two different values for that field across the given configuration segments. Thus, a parameter error is identified in the metatemplate by a field that contains a parameter instead of a constant. In our example, out of the 104 ACLs that contain line 6, 94 use the mask 255 for parameter A, while 10 of them use the mask 127. The network engineers confirmed that the 10 ACLs are erroneous, permitting more traffic than intended.

SELFSTARTER is useful for finding errors and inconsistencies in networks that are managed through manual creation of individual node configurations. Perhaps surprisingly, we have also found SELFSTARTER to be useful for networks that employ forms of automation to manage configurations.

First, many network engineers employ *configuration templates*, with one parameterized template per role in the network. This simplifies network management since node configurations can be created by instantiating the relevant template with node-specific parameter values. For example, the university network described above has a template for all building routers. However, this network still suffered from multiple misconfigurations identified by SELFSTARTER. The problem is that node configurations tend to drift over time from their original templates. Such *template drift* happens for several reasons. First, operators often must manually edit a node's configuration, for example to quickly address a problem or to perform maintenance of various kinds. Second, the templates themselves get updated over time, and often operators are relied upon to manually perform the necessary configuration updates. Hence there is typically still considerable manual configuration in practice, which can easily lead to errors.

Second, some networks employ automated scripts to deploy, update, and validate configurations consistently. In this situation, SELFSTARTER is useful to protect against bugs in the automation software itself. Automation may actually *increase* the need for validation tools like SELFSTARTER, since small errors in the automation can lead to large, network-wide misconfigurations. The wide-area network (WAN) that we analyzed employs automation scripts for various purposes, and SELFSTARTER discovered misconfigurations that were due to previously unknown script errors. Once reported, the network engineers promptly confirmed and fixed the errors.

3 Structured Generalization

Our *structured generalization* algorithm takes as input a collection of configuration segments and outputs a metatemplate for these segments in the form shown in Figure 1. After an overview of the key challenges, we present the generic algorithm and then describe how it is instantiated for ACLs, prefix lists, and route policies.

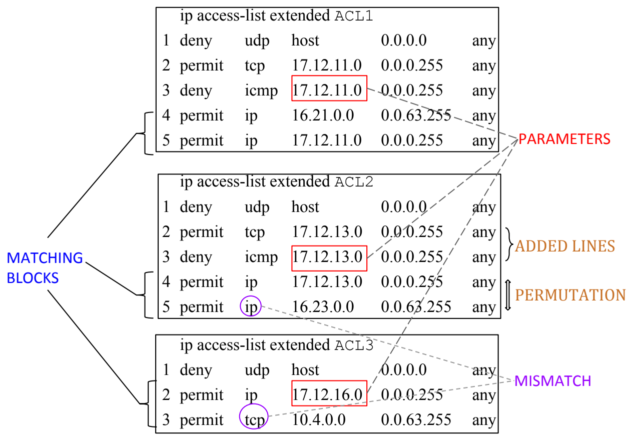
To provide high-quality outliers, the algorithm must be given configuration segments that are intended to be similarly structured. For example, it is common for routers to be partitioned into *roles* (e.g., border routers, core routers) and for the routers within a role to be configured similarly. We rely on the user to provide an appropriate set of configuration segments to be templated. In our experience (see §6), operators know the roles in their network and can quickly identify the segments that are intended to be similar, so this requirement does not pose a large burden in practice.

3.1 Challenges

Suppose that we wish to create a metatemplate for the three different configurations of the same ACL shown in Figure 2. We use this example to illustrate the challenges that our algorithm must address.

Consider the first two ACLs in Figure 2. Their first lines are identical, so clearly they should be matched to one another. However, their second lines differ — they apply to different source IP addresses. A naive approach is to simply not match these lines to one another. However, such an exact-matching criterion is much too strong, as it is common for corresponding ACLs to be similar but not identical across nodes, for example to treat local addresses specially. Therefore, the algorithm must be able to match non-identical lines to one another. This requirement is met using *parameterization*. In this example, we can introduce a parameter to represent the third octet in the source IP address, indicating that this octet differs in each ACL while the rest of the line is identical.

While limited parameterization is necessary, arbitrary parameterization would yield undesirable results. For example, it would not be useful if the metatemplate matches a `permit`



```
  ip access-list extended ACL1
1 deny    udp   host        0.0.0.0      any
2 permit  tcp   17.12.11.0  0.0.0.255    any
3 deny    icmp  17.12.11.0  0.0.0.255    any
4 permit  ip    16.21.0.0   0.0.63.255   any
5 permit  ip    17.12.11.0  0.0.0.255    any

  ip access-list extended ACL2
1 deny    udp   host        0.0.0.0      any
2 permit  tcp   17.12.13.0  0.0.0.255    any
3 deny    icmp  17.12.13.0  0.0.0.255    any
4 permit  ip    17.12.13.0  0.0.0.255    any
5 permit  ip    16.23.0.0   0.0.63.255   any

  ip access-list extended ACL3
1 deny    udp   host        0.0.0.0      any
2 permit  ip    17.12.16.0  0.0.0.255    any
3 permit  tcp   10.4.0.0    0.0.63.255   any
```

Figure 2: Configurations of the ACLs matching the `ACL*` regex from three different routers highlighting the challenges.

line with a `deny` line, as their behavior is not at all similar. As a more subtle example, consider the last line in `ACL3` in Figure 2. This line is similar to the last line in `ACL2`. However, since the two lines specify different protocols (tcp vs. ip), it is unlikely that they are intended to serve the same function in the ACLs. Hence, despite being similar, it may not make sense to match these two lines. Therefore, there is a need for an ability to specify different constraints on what can and cannot be parameterized for different segment types.

Now consider lines 4 and 5 in `ACL1` and `ACL2`. Line 4 in `ACL1` is similar to line 5 in `ACL2`. To match these lines to one another, the algorithm must support *reordering* of configuration lines. Doing so requires a scoring metric to determine which line pairs are the best matches. Such a scoring metric also naturally handles missing lines, as in `ACL3`, resulting in there being multiple options for how to match the lines present in `ACL3` to those in the other ACLs.

As with parameterization, allowing arbitrary line reordering would yield undesirable results. Specifically, reordering a `permit` line and a `deny` line can, in general, change the semantics of an ACL. Therefore, matching lines across ACLs in a way that requires such a reordering can potentially mask important differences between ACLs. Thus, as with parameterization, we need the ability to specify constraints on the allowable reorderings for different segment types.

Our structured generalization algorithm, described below, meets these challenges through a novel two-level approach.

3.2 Algorithm

The algorithm partitions segments into *blocks* of lines that must not be reordered with one another and hence are matched using sequence alignment. However, when matching two blocks with one another, the algorithm employs bipartite matching on their lines, thereby supporting line reordering within blocks and also inducing parameters wherever two

Algorithm 1 STRUCTURED GENERALIZATION

```
Input: $S_1, S_2, \ldots S_n$ - Sequence of segments to be templated
Output: Metatemplate for $S_1, S_2, \ldots S_n$
1: Metatemplate $T \leftarrow S_1$
2: for $S_i \leftarrow \{S_2, \ldots, S_n\}$ do
3:     Block sequence $B_1 \leftarrow$ GETBLOCKSEQUENCE($T$)
4:     Block sequence $B_2 \leftarrow$ GETBLOCKSEQUENCE($S_i$)
5:     Alignment $A \leftarrow$ ALIGNSEQUENCES($B_1$, $B_2$, MISMATCHSCORE)
6:     $T \leftarrow$ GENERATEMETATEMPLATE($A$, $B_1$, $B_2$)
7: end for
8: $T \leftarrow$ MINIMIZEPARAMETERS($T$)
9: return $T$
```

Algorithm 2 MISMATCHSCORE

```
Input: $\{b_1, b_2\}$ - Blocks to be matched
Output: A score for aligning $b_1, b_2$ and a matching between their lines
1: Line sequence $L_1 \leftarrow$ GETLINESEQUENCE($b_1$)
2: Line sequence $L_2 \leftarrow$ GETLINESEQUENCE($b_2$)
3: Score $S$, Matching $M \leftarrow$ MINIMUMWEIGHTBIPARTITEMATCHING($L_1$,
   $L_2$, LINESCORE)
4: return $S$, $M$
```

matched lines differ. The amount of reordering and parameterization are respectively controlled through a function that partitions segments into blocks and a function that determines edge weights for bipartite matching.

The structured generalization algorithm is shown in Algorithm 1. The algorithm starts by treating the first segment S_1 as the initial metatemplate (Line 1). It then iteratively combines the current metatemplate with each remaining configuration segment, one at a time, to produce the final metatemplate.

At each iteration the algorithm performs three main steps:

1. Block Generation: A *block* is a contiguous sequence of lines within a segment that can be reordered with one another. The GETBLOCKSEQUENCE function partitions each segment into blocks (Lines 3 to 4) and is specific to a particular segment type. For example, for ACLs the GETBLOCKSEQUENCE function partitions a segment into maximal sequences of lines that have the same action (`permit` or `deny`). The block abstraction and corresponding GETBLOCKSEQUENCE function provide a natural way to admit reorderings within a block while forbidding those across blocks.

2. Block Alignment and Line Matching: Blocks are matched in the two block sequences B_1 and B_2. To prevent cross-block reorderings, a standard *sequence alignment* algorithm is employed at the block level (Line 5). Such an algorithm finds a minimum-cost alignment between the two block sequences, allowing gaps but not reorderings.

To meet the need to support within-block reorderings, our algorithm leverages the fact that the sequence alignment algorithm requires a function MISMATCHSCORE that provides the cost of matching two blocks to one another.[2] The MISMATCHSCORE function is shown in Algorithm 2. The function first breaks the two blocks into lines, and then it performs a minimum-weight bipartite matching between the two line sequences using a standard matching algorithm, thereby supporting line reordering. The function returns the score for the best matching and also the matching itself.

The MISMATCHSCORE function uses another function, LINESCORE, to assign weights to each edge in the bipartite graph (between a pair of lines). This function is specific to each type of segment and provides a flexible way to prevent undesired line matchings and to prioritize among different possible line matchings.

3. Parameterization: Once the two block sequences have been aligned, the metatemplate is generated (Line 6). As described above, for each pair of blocks that are aligned with one another, the MISMATCHSCORE function also provides a matching at the line level. For each pair l_1 and l_2 of matched lines, the metatemplate includes their *least general generalization* (lgg) [34]. Intuitively, this is simply a version of l_1 where a parameter is introduced in each field where it differs from l_2. For example, the lgg of the second lines in the first two ACLs in Figure 1 is `permit tcp 17.12.P.0 0.0.0.255 any`, where `P` is a new parameter.

As a final step, the number of parameters in the metatemplate is globally minimized (Line 8). Specifically, let $P(S)$ denote the value that parameter P takes in segment S. If two parameters P_0 and P_1 in the metatemplate have the property that $P_0(S_i) = P_1(S_i)$ for each segment S_i, then the parameters are merged into a single parameter.

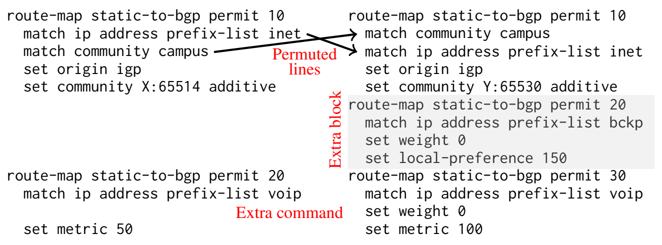
3.3 Instantiation for ACLs and Prefix Lists

This subsection discusses how the structured generalization algorithm is instantiated for ACLs and prefix lists, which are handled identically. Doing so requires providing specific GETBLOCKSEQUENCE and LINESCORE functions to the generic algorithm above. The goal is to take advantage of the particular properties of ACLs and prefix lists to both allow for flexible templating and ensure that generated templates are *actionable*, i.e., they facilitate the identification of common configuration errors.

The GETBLOCKSEQUENCE function partitions an ACL or prefix list into maximal sequences of lines that have the same action (`permit` or `deny`). This allows reorderings guaranteed to have no behavioral effect. The notion of blocks can be relaxed to admit more reorderings. For example, reordering a `permit` line with a `deny` line does not change behavior when the sets of packets that they handle are disjoint. However, in practice, we have not encountered the need to support such reorderings, so the extra complexity is not warranted.

The LINESCORE function for ACLs and prefix lists prohibits matching lines that differ in either their action (`permit` or `deny`) or protocol by returning an infinite weight. All other differences (e.g., in source or destination IP addresses) are

[2] Sequence alignment algorithms also require a function to provide the cost of introducing a gap; we use a simple metric for this cost based on the size of the unmatched block.



```
route-map static-to-bgp permit 10
  match ip address prefix-list inet
  match community campus
  set origin igp
  set community X:65514 additive




route-map static-to-bgp permit 20
  match ip address prefix-list voip

  set metric 50
```

```
route-map static-to-bgp permit 10
  match community campus
  match ip address prefix-list inet
  set origin igp
  set community Y:65530 additive
route-map static-to-bgp permit 20
  match ip address prefix-list bckp
  set weight 0
  set local-preference 150
route-map static-to-bgp permit 30
  match ip address prefix-list voip
  set weight 0
  set metric 100
```

Figure 3: Example: Two route policies.

allowed but are penalized by an increase in overall score. Specifically, consider some field of the two lines that is allowed to differ (e.g., the first octet of the source IP address). If the lines agree on the value of this field, then the overall score is unchanged. If one field contains a constant while the other contains a parameter name (which must have been introduced in an earlier iteration), then the overall score is increased by 1, since an exact match is preferable. Finally, if the lines contain different constant values in this field, then the overall score is increased by 2, since a new parameter must be introduced in order to match the lines, so either of the above two cases is preferable. Despite its simplicity, this scoring function works well across all of our experiments (§6).

§4 contains a detailed example of the structured generalization algorithm applied to the ACLs in Figure 2.

3.4 Instantiation for Route Policies

Route policies are flexible configuration segments used in several contexts, including route redistribution filtering, policy-based routing, and BGP policy implementation. A route policy is defined as a sequence of *route map clauses*, each of which contains an action (e.g., `permit`), a list of `match` lines, and a list of `set` lines.[3] The `match` lines provide a flexible way to select route announcements of interest based on the route attributes. The `set` lines then update matching announcements (e.g., to add a particular community tag). Figure 3 shows two example route policies.

The GETBLOCKSEQUENCE function for route policies puts each clause in one block, so the left and right route policies in Figure 3 respectively contain two and three blocks. This allows the `match` lines within a route map to be reordered with respect to one another, and similarly for the `set` lines.[4] Since all `match` lines have to succeed for a route announcement to match the route map, reordering them has no effect. In principle, reordering `set` lines can change behavior, specifically if one `set` line reads a value that was updated by a previous `set` line. In our experience, however, order dependence is rare: we have encountered only one such situation (where the deletion to a community attribute was followed by an addition). Therefore we opted to allow reordering of `set` lines within route maps to handle the common case properly.

[3]Here we have used the syntax from Cisco IOS; the JunOS syntax from Juniper uses different keywords but is semantically similar.

[4]Technically it also allows `match` and `set` lines to be reordered with one another, but that is not syntactically legal so will never arise.

The LINESCORE function for route policies allows two lines to be matched in the bipartite matching only if they refer to the same *attribute* of the route announcement. For example, the `match community campus` line in the left route policy of Figure 3 can only match against other `match community` lines, and the following `set origin igp` line can only match against other `set origin` lines.

With route policies it is common for `match` lines to refer to prefix lists and access lists defined elsewhere in the configuration. For example, the first `match` line in the left route policy refers to the prefix list named `inet`. When scoring two lines that refer to a named segment, we choose to perform a *shallow* comparison that considers them to exactly match if they refer to same-named segments. An alternative would be to perform a *deep* analysis that ignores the names and instead expands the definitions of these segments in order to recursively compare them. From our experiments on a large cloud provider network, we found that simple name comparison works well, since the network names segments identically across configurations. Further, if two same-named segments do differ in their definitions, that will be caught during metatemplating of those segments.

Figure 3 illustrates the best alignment of the two route policies, based on the GETBLOCKSEQUENCE and LINESCORE functions described above. Specifically, the second clause of the left policy is aligned with the third clause of the right policy. That is the lowest-cost alignment since these route maps match on the same-named prefix list announce and both have a line to set the `metric`.

4 A Templating Example

This section describes how Algorithms 1 and 2 generate a metatemplate for the three ACLs in Figure 2. Figure 4 shows `ACL1` and `ACL2` side by side; we use superscripts to uniquely refer to each `permit` and `deny` line. The STRUCTURED GENERALIZATION algorithm designates `ACL1` as the initial metatemplate (Line 1) and iteratively incorporates `ACL2` and `ACL3` to produce the final metatemplate (Lines 2 to 7).

In the first iteration, the "Block Generation" step partitions `ACL1` (Line 3) and `ACL2` (Line 4) into blocks. For `ACL1`, the algorithm generates the block sequence B_1 consisting of four blocks — $\mathsf{D_a}$, $\mathsf{P_a}$, $\mathsf{D_b}$, and $\mathsf{P_b}$ — where $\mathsf{D_a}$ contains deny^1, $\mathsf{P_a}$ contains permit^1, $\mathsf{D_b}$ contains deny^2, and $\mathsf{P_b}$ contains permit^2 and permit^3. The algorithm generates an analogous block sequence B_2 for `ACL2`, with four blocks that we denote $\mathsf{D_x}$, $\mathsf{P_x}$, $\mathsf{D_y}$ and $\mathsf{P_y}$.

Next, in the "Block Alignment and Line Matching" step, the algorithm uses the sequence alignment algorithm, which in turn relies on the MISMATCHSCORE(b_1, b_2) function in Algorithm 2. It turns out that the following alignment A is the

```
ip access-list extended ACL1                  ip access-list extended ACL2
 deny¹   udp  host        0.0.0.0     any      deny³   udp  host        0.0.0.0     any
 permit¹ tcp  17.12.11.0 0.0.0.255   any      permit⁴ tcp  17.12.13.0 0.0.0.255   any
 deny²   icmp 17.12.11.0 0.0.0.255   any      deny⁴   icmp 17.12.13.0 0.0.0.255   any
 permit² ip   16.21.0.0  0.0.63.255  any      permit⁵ ip   17.12.13.0 0.0.0.255   any
 permit³ ip   17.12.11.0 0.0.0.255   any      permit⁶ ip   16.23.0.0  0.0.63.255  any
```

Figure 4: Side by side comparison of ACL1 and ACL2

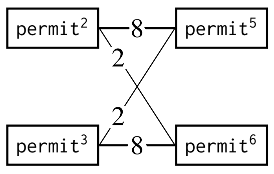


Figure 5: Bipartite graph of P_b and P_y

```
ip access-list extended ACL*                      ip access-list extended ACL3
 deny         udp  host      0.0.0.0     any       deny     udp  host        0.0.0.0      any
 permit       tcp  17.12.A.0 0.0.0.255   any
 deny         icmp 17.12.B.0 0.0.0.255   any
 permit³,⁵    ip   17.12.C.0 0.0.0.255   any       permit⁷  ip   17.12.16.0  0.0.0.255    any
 permit²,⁶    ip   16.D.0.0  0.0.63.255  any       permit⁸  tcp  10.4.0.0    0.0.63.255   any
```

Figure 6: Comparing the metatemplate for ACL1 and ACL2 with ACL3

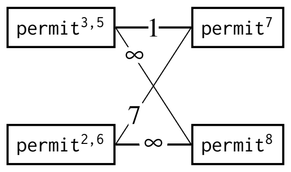


Figure 7: Bipartite graph of last permit block

optimal alignment for the two sequences.

$$(D_a, P_a, D_b, P_b)$$
$$(D_x, P_x, D_y, P_y)$$

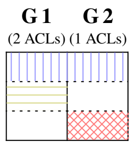
The single line in D_a is matched with the single line in D_x, and similarly for the next two pairs of blocks in the alignment. For the last pair of blocks (P_b, P_y), the algorithm constructs the bipartite graph shown in Figure 5 to determine line matchings. The LINESCORE(l_1, l_2) function assigns the edge weight of permit^2 and permit^6 as 2 since the lines differ in the second octet of the source IP, but the edge weight of permit^2 and permit^5 is assigned as 8 since they differ in four octets total across the source IP and source mask. Therefore, the minimum weight matching (Line 3) matches permit^2 with permit^6 and permit^3 and permit^5, thereby performing the allowable reordering.

The "Parameterization" step generates the metatemplate by introducing parameters based on the produced sequence alignment and line matchings (Line 6). The resultant metatemplate of ACL1 and ACL2 is shown on the left side in Figure 6. The two ACLs have a common template; the line produced by merging permit^3 with permit^5 is shown as permit^{3,5} in the metatemplate, and similarly for permit^{2,6}.

The next iteration incorporates ACL3. The metatemplate from the first iteration and ACL3 are both partitioned into blocks, and then these blocks are aligned in the same way as described above. The best block alignment and line matching are shown in Figure 6. As an example, the bipartite graph constructed to calculate the MISMATCHSCORE of the last block in each block sequence is shown in Figure 7. The edge weight of permit^{3,5} and permit^7 is 1 since there is already a parameter in the metatemplate for the source IP address octet where the two lines disagree. The LINESCORE function gives the edge between permit^{3,5} and permit^8 a score of ∞ since lines with different protocols cannot be matched. The minimal-weight matching matches permit^{3,5} with permit^7 and leaves lines (permit^{2,6} and permit^8) unmatched. This

```
ip access-list extended ACL*
 deny        udp  host      0.0.0.0     any
 permit      tcp  17.12.A.0 0.0.0.255   any
 deny        icmp 17.12.B.0 0.0.0.255   any
 permit³,⁵,⁷ ip   17.12.C.0 0.0.0.255   any
 permit²,⁶   ip   16.D.0.0  0.0.63.255  any
 permit⁸     tcp  10.4.0.0  0.0.63.255  any
```

(a) Metatemplate of all three ACLs



(b) Groups

```
ACL1: [A = 11, B = 11, C = 11, D = 21]
ACL2: [A = 13, B = 13, C = 13, D = 23]
ACL3: [C = 16]
```

(c) Router parameter value map

Figure 8: Result of templating ACL1, ACL2 and ACL3

example demonstrates the need to match blocks but also to identify gaps due to missing/extra lines.

The resultant metatemplate for the three ACLs is shown in Figure 8(a), the ACL groups are shown in Figure 8(b), and the parameter-value map is shown in Figure 8(c). Finally, as explained in §3.2, the algorithm can create more parameters than necessary. For example, in Figure 8, parameters A, B and C are redundant: for every router *R* that requires all of these parameters, *R* instantiates the parameters with the same value. Therefore, the parameter minimization step (Line 8) merges A, B and C into a single parameter.[5]

5 Implementation

SELFSTARTER[6] takes as input a set of router configurations and a segment name or regular expression. It outputs a

[5] A degenerate case occurs when two parameters are never required together by any router; in that case we do not merge the parameters since they are likely to be logically unrelated.

[6] https://github.com/SivaKesava1/SelfStarter

metatemplate for all segments that match the given name or regular expression, a partitioning of each segment into groups that share a common template, and a parameter mapping for each segment, as shown in Figure 8.

SELFSTARTER is written in Python. It uses PYBATFISH,[7] a client for the BATFISH network configuration analysis tool [16], to parse the raw vendor-specific configuration files into BATFISH's vendor-agnostic format. This format provides a structured representation of the configuration data, and our algorithm works directly on this representation. BATFISH can parse many different configuration formats, including those from Cisco, Juniper, and Arista, so in turn SELFSTARTER can infer templates for segments from all of these vendors.

Our structured generalization algorithm uses a standard algorithm for sequence alignment based on dynamic programming, in order to align block sequences. Lines within a block are matched using the Python library `munkres`, which implements Munkres' improvement on the standard Hungarian matching algorithm to be (strongly) polynomial [25, 26, 33]. We also perform one major optimization over the algorithm. Given the collection of segments to template, SELFSTARTER first removes all duplicates from the collection — segments whose representation in BATFISH's format are identical to that of some existing segment in the collection. Since there are often subsets of the configurations in a role that are meant to be exactly identical, this optimization can significantly improve the efficiency of structured generalization by reducing the number of calls to the expensive matching algorithms.

The order in which segments are considered for templating can affect the final metatemplate and groups produced. Our implementation uses a simple heuristic. The segments are partitioned based on their line counts. The segments in the largest partition (i.e., the partition containing the most segments) are templated first, choosing segments randomly from that partition until it is empty. The remaining segment partitions are then processed in order of decreasing partition size. Intuitively this heuristic tries to template segments that are likely to be outliers last, so that their impact on the templates of other segments is minimized.

The structured generalization algorithm produces a metatemplate in BATFISH's vendor-agnostic format, but SELFSTARTER must output the metatemplate in some vendor-specific format in order to be understandable to network engineers. Currently SELFSTARTER outputs metatemplates in Cisco's IOS format. This has been sufficient to get feedback for our experiments (§6), since the majority of nodes use that format, and it is similar enough to the other formats for the engineers to understand the results. It would be straightforward to add pretty printers to output the metatemplate in other vendor-specific formats in the future.

Given the results of the structured generalization algorithm, SELFSTARTER finally produces the output visualization as shown in Figure 8. First the metatemplate lines are partitioned, where two lines are placed in the same subset if and only if for every segment, either both lines are in the segment or neither is. Then a color is chosen for each subset, each inferred template is mapped to the set of colors of its lines, and the colored tables are created and output as HTML files.

[7] `https://github.com/batfish/pybatfish`

6 Evaluation

To evaluate SELFSTARTER, we applied it to a collection of datacenter networks from a large cloud provider, a wide area network from the same cloud provider, and the campus network of a large university. These networks differ widely in their structure, scale, and management style. Yet SELFSTARTER identified misconfigurations in all of them; in two of the three networks SELFSTARTER identified previously unknown errors. Further, SELFSTARTER's inferred templates closely match any manually written templates that exist for these networks. A dominant cause of the errors that SELFSTARTER identified were planned configuration updates not yet applied or inconsistently applied. We ran SELFSTARTER on a 3.6 GHz quad-core machine with 32 GB of RAM.

Methodology: For each network, we obtained a snapshot of the router configurations and partitioned them into roles based on router names, sometimes with the help of the network operator. We then parsed the configurations with BATFISH and ran SELFSTARTER on the output once per *triple*, where a triple is a tuple of a segment type (e.g., ACL), role (e.g., the border routers), and segment name or regular expression (e.g., `border-out-1`). SELFSTARTER produces a metatemplate for each triple.

Network	Number of routers	Number of roles	Number of triples	BATFISH parsing time	SELFSTARTER running time
Datacenter	$O(10000)$	$O(1000)$	25475	150 min	90 min
WAN	$O(100)$	$O(100)$	21011	14 min	20 min
Campus	106	1	6	2 min	<2 min

Table 1: Networks' parsing and running times

Table 1 shows the order of magnitude of each network in number of routers and roles, along with the number of triples on which we ran SELFSTARTER[8] and the time for BATFISH parsing as well as the total time to run on all triples.

We define a **consistent triple** to be one whose metatemplate consists of only a single group (i.e., all segments meet the same template). Similarly, an **inconsistent triple** is one for which SELFSTARTER generates a metatemplate with at least two groups. We consider *all* inconsistent triples to be group outliers, and we report the number of such outliers as well as the number that are true/false positives, by comparing with the ground truth (either a "golden" configuration or the

[8] For confidentiality, we cannot disclose the exact number of routers and roles for the cloud provider.

network operator). Perhaps surprisingly, this very coarse way of identifying group outliers is quite effective in identifying real errors in practice, as we show below.

We tried to automatically identify parameter outliers, where a parameter value is considered an outlier if the value's frequency is below a threshold. We considered different threshold functions – $X\%$ of max frequency, $X\%$ of average frequency, etc. — and used an existing technique [28] to identify a good threshold value by finding the point at which the number of outliers spikes as X is varied. However, in the cloud provider networks nearly all parameter outliers that we investigated using this approach were false positives. The global nature of the cloud provider networks makes it likely that there will be many different parameter values, including some used infrequently. Therefore, our experiments do not identify parameter outliers for the cloud provider networks. For the university network, the number of metatemplates and parameters was small enough for us to manually identify parameter outliers and then validate them with the network operators.

6.1 Datacenter Networks

We applied SELFSTARTER to a large collection of production datacenter networks, totaling tens of thousands of devices and hundreds of millions of lines of configuration, from a cloud provider. The datacenters are set up in a folded Clos topology and run the eBGP routing protocol with private AS numbers, as described in an RFC [27]. We obtained a snapshot of device configurations from December 2018.

Ground truth: The configuration files for all network devices are generated automatically from a set of hand-written templates, which are kept up-to-date as "golden" templates. The templates are parameterized, and a separate database maintains the parameter values for each node (e.g., its list of SNMP monitoring servers). Both the golden templates and the parameters database are subject to periodic updates. A software service automatically generates new per-device "golden" configuration based on these updates and installs them on the running devices. However, there are constraints on when and how often it is appropriate to update a device. For example, a device that is a single point of failure must be updated only after customers that could be impacted are safely transitioned. The software service takes these constraints into account when updating the configurations.

We consider an inconsistent triple to be a true positive if the configurations of at least one group of nodes that SELFSTARTER identifies differ from their golden configurations. Because of the software service that automates configuration updates, we did not expect to (and indeed did not) find new errors with SELFSTARTER. Rather, this experiment allows us to objectively validate SELFSTARTER by comparing its results with a well-maintained source of ground truth.[9]

[9]However, in principle SELFSTARTER can still be useful for this network, to catch errors in the automation service itself.

Triples: Recall that a triple contains a segment type, role, and segment name. Since each datacenter network defines ACLs, prefix-lists, and route-policies, we include all three segment types in our triples. Each node's name includes both the name of the datacenter to which it belongs and its *tier* within the datacenter (e.g., top-of-rack); we treat each (datacenter, tier) pair as a unique role. Finally, for each segment type and role, we create a triple for each unique segment name of that type in the role, resulting in more than 20,000 triples.

Segment Type	Consistent Triples	Inconsistent Triples		
		Reported	Investigated	True positives
ACLs	9700	938	400	400
Prefix Lists	2954	0	-	-
Route Policies	11653	230	230	230

Table 2: Datacenter Results

Results: SELFSTARTER identified 1168 group outliers across the three segment types (Table 2), which is fewer than 5% of all triples. We investigated all 230 of the route-policy outliers. We randomly selected 400 ACL outliers to investigate while ensuring that this subset contains at least one triple for each different segment name that appears in the set of outliers. All 630 outliers were determined to be true positives. Specifically, in all cases, at least one of the groups of routers was correctly following the golden template and the difference between configurations was due to a configuration update had been applied to some, but not all, of the nodes in the same role. As mentioned earlier, updates are delayed for many reasons, so such differences are expected and are eventually resolved by the software service.

6.2 Wide Area Network

The WAN we analyzed is one of the largest backbone networks in the world; it interconnects North America, South America, Asia, Europe, Africa, and Oceania. The backbone consists of hundreds of thousands of kilometers of fiber, hundreds of routers, and millions of lines of router configuration. The configurations for the routers are stored in a centralized database, from which we obtained a June 2019 snapshot. All routers are JunOS-based and use the flat-juniper configuration language format.

Ground truth: The WAN does not employ explicit configuration templates. However, the network operators rely partly on scripts to manage the network; the templates are implicit in these scripts. Typically, each script performs one specific task. As an example, a script configures a set of route policies and verifies that they are consistent across a set of devices. In effect, the equivalent of "golden" configurations described for datacenters do not exist for the wide area. Thus, we asked the network operators to help classify SELFSTARTER's outliers as true/false positives.

Triples: We determined useful network roles using an iterative process. We initially divided nodes into roles based on only their network functionality from their names — edge routers, border routers, core routers, route reflector routers and so on. As in the datacenter, we then created one triple for each unique segment name, per segment type and role.

The initial results using this role scheme contained numerous false positives. Upon consulting a network operator, we immediately (within minutes) received the feedback that these roles were too coarse-grained as well as guidance on how to further refine the roles. The first refinement was to take into account the operating environment of each node. For example, two nodes, where one has external peering enabled and the other does not, should not be considered to be in the same role. Fortunately, we were able to utilize information in a file, maintained by the operator, that lists the operating environment of each node. The second refinement was to take into account geographic location. Routers have certain specializations based on geographical regions in order to meet local policies such as government-specific privacy requirements.

In the end, the number of routers is only roughly 5$\times$ the number of roles. In retrospect this makes sense for the WAN due to its global spread. Devices necessarily have many policy differences across regions, for instance based on local peering relationships. Though our initial guess of roles was overly coarse, it was trivial for the operator to immediately identify the issue and provide refined role information.

The WAN defines ACLs, prefix lists, and route policies. However, to date we only have feedback from the operators about SELFSTARTER's output for prefix lists and route policies; thus we omit ACLs from the results below.

Segment Type	Consistent Triples	Inconsistent Triples		
		Reported	Investigated	True positives
Prefix Lists	10042	166	138	7
Route Policies	10969	56	33	33

Table 3: WAN Results

Results: SELFSTARTER identified 222 inconsistent triples (Table 3), which is $\sim$1% of all triples. All outliers that were flagged as true positives were previously unknown and have since been remediated by the operators. All 33 investigated triples for route policies were verified as true positives by the network operators. There were several different root causes. Interestingly, one class of errors was due to bugs in the automation scripts. Specifically, a script that checks for the existence of certain commands in a route policy accidentally did not consider the order of these commands. Since SELFSTARTER takes order into account, its metatemplate contained multiple groups and hence identified ordering errors that change the behavior and that the script missed. Another class of errors, which caused five inconsistent triples, was due to a spurious community tag being added to some route announcements in a few routers.

SELFSTARTER was much less effective at finding real errors in prefix lists, with only 7 out of 138 investigated triples determined to be true positives. In 95 out of 138 triples, the Juniper command `apply-path` is used to create a prefix list by expanding an existing set of addresses defined elsewhere in the configuration, such as the set of local IPs or NTP servers. The seven true positives represented cases where there were inconsistent sets of loopback interfaces defined on different routers, and required cleanup. For the other 88 cases, the operator informed us that it is expected that every router's prefix list will expand to a different set of IPs, and that they can have different numbers of such addresses. For the remaining prefix-lists, the operator informed us that those prefix-lists are locally significant on every router and will always be different. Since SELFSTARTER creates multiple groups whenever two segments have different numbers of lines, this led it to report these spurious inconsistent triples.

On the positive side, it was easy for the operator to quickly understand SELFSTARTER's output and identify these cases as false positives. In fact, the total operator time to classify all of SELFSTARTER's results, for both prefix lists and route policies, was under 30 minutes. Going forward, it would be simple to allow operators to suppress errors reported for segments/roles with well-known differences.

6.3 University Network

The university network consists of approximately 1000 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 obtained a snapshot from May 2019 of the network configurations.

Ground truth: Configurations in the network were created using a mix of manual setup and templates. However, even where templates exist, the configurations are still updated directly over time to meet evolving policy needs, and the original templates are not always kept up to date. Therefore, we asked the network engineers to validate our results.

Triples: Node names include a two-letter abbreviation to indicate the network role, for example *cr* for core routers and *br* for building routers. We analyzed only the 106 building routers in the network. This network role was chosen because it is the most interesting one for our purposes – it is the only role that contains many segments that are intended to be similar to one another. Specifically, the role contains six distinct ACLs that appear in nearly all routers. ACLs account for more than one third of the lines in these configurations. The building routers do not contain prefix lists or route policies.

We ran SELFSTARTER with a regular expression for each ACL rather than an exact name, since they have slightly different names in each router. Specifically, the name of each router's version of an ACL includes the router's associated building name and the date on which the ACL was created.

ACL Pattern	Number of ACLs in largest group	Number of ACLs in other groups
br_aux_mgmt_*_in_*	88	18
br_mgmt_*_in_*	82	24
br_wlan_mgmt_*_in_*	80	24
br_aux_mgmt_*_out_*	84	22
voip_*_in_*	61	22
voip_*_out_*	61	22

Table 4: May 2019 "br" router ACL results.

Results: Table 4 summarizes the results. For each ACL SELFSTARTER identified one dominant group of nodes that share a common template, along with one or more smaller groups of nodes that have a different template. Hence non-dominant groups potentially indicate misconfigurations.

To date the network engineers have provided feedback on the first three ACLs in the table. In all three cases the network engineers have confirmed that the group outliers that SELFSTARTER identified are indeed misconfigurations: every ACL that SELFSTARTER placed in a non-majority group has at least one misconfiguration.

For example, the metatemplate and the groups that SELFSTARTER inferred for the first ACL regex in Table 4 was shown in Figure 1. The 18 ACLs in Groups 2 and 3 include some old `deny` lines. Initially, IP spaces 14.10.49.0/24, 14.10.50.0/24, and 15.8.228.0/20 were allocated for infrastructure management; thus access to them was restricted using `deny` lines (ACL Lines 3 – 5). However, that allocation changed at some point to 14.10.0.0/19 and 17.7.240.0/20; thus the intent was to deny traffic to these new IP spaces (ACL Lines 1 – 2) and remove the old `deny` lines. Because it is each department's responsibility to update the ACLs for their building routers, some of the ACLs were not properly updated. The second and third ACLs in Table 4 exhibited similar misconfigurations, all of which were confirmed by the network engineers.

The network engineers have templates for these ACLs and so we manually compared the template that SELFSTARTER inferred for the dominant group with those templates for the first three ACLs in Table 4. For the first two ACLs, SELFSTARTER's template matches the network's hand-written templates. Specifically, the templates are *line-for-line identical*, except that in some cases SELFSTARTER's template has a concrete value where the golden template has a parameter. For example, SELFSTARTER might learn that a particular line uses IP address `1.2.3.A`, since all segments agree on the first three octet values, but the hand-written template treats the entire IP address as a parameter.

For the third ACL, SELFSTARTER's template for the dominant group does not match the network's template. Specifically, the network's template has two additional `permit` lines (one based on IP and another on ICMP). The network engineers informed us that this template was indeed stale. This shows that a possible use case for SELFSTARTER is to automatically identify "template drift".

Finally, we manually identified three parameter outliers in the first ACL. Earlier we said that parameter `A` in line 6 (ACL Line 6) of the metatemplate in Figure 1(a) was confirmed as a misconfiguration. We also asked the network engineers about parameters `B` and `E` (line 7). Both of these turned out to be intentional differences. For example, in the case of parameter `E` two building routers required more hosts than 255 and so were allocated a larger IP space than the other routers.

7 Discussion

Our experience applying SELFSTARTER to real-world networks and interacting with the operators has led to several observations, which we discuss here.

SELFSTARTER assumes that many router configurations in a network are *structurally similar* to one another. Our experiments largely bear out this assumption, since network operators typically employ configuration templates or common guidelines to simplify the configuration of groups of nodes. However, when this assumption does not hold, for example in the prefix lists of the WAN that we analyzed, then SELFSTARTER will not be as useful.

Our experimental evaluation indicates that SELFSTARTER can be useful in different kinds of networks, which are managed in different ways. Even where templates exist, they can become stale over time as the running configurations are updated. Even where automation exists, the automation can be incomplete or itself be a source of errors. Because SELFSTARTER takes as input only the final per-node configurations, it provides a useful form of redundancy for validating configurations, regardless of how they were created.

Finally, SELFSTARTER's structural approach to outlier detection means that it cannot determine the *behavioral* differences between outliers and non-outliers. However, in our experience a key advantage of SELFSTARTER is that its results are very easy for operators to understand, precisely because a metatemplate retains the structure of the original configuration segments. All of the network operators with whom we interacted were able to quickly decide whether an identified outlier was a true or false positive.

8 Related Work

To our knowledge, ours is the first technique to automatically infer templates for network configuration segments. We use these templates to identify misconfigurations, so we compare against other techniques for doing so.

Network Verification Network verification for the data plane (e.g., [6,11,23,24,29,31,39]) and control plane (e.g., [7, 14–16,18]) models the *semantics* of the network, which allows them to verify deep behavioral properties. However, they rely on the user providing a formal specification, and otherwise are limited to checking generic properties like the absence of loops. In contrast, SELFSTARTER leverages the *structural* similarity of many router configurations to identify network-specific errors without a specification, but SELFSTARTER cannot map these errors to specific undesired behaviors.

Minesweeper [7] can verify *functional equivalence* of two router configurations, which could be used to identify outliers. However, exact functional equivalence is much too strong a criterion in general and so would lead to a high false positive rate and make it harder to spot errors. For example, partitioning the 88 ACLs of Group 1 from Figure 1, all of which are correct, by functional equivalence would result in 44 groups, each of size 2. On the plus side, by modeling a segment's behavior Minesweeper can safely allow reorderings that SELFSTARTER would spuriously flag, for example swapping the order of unrelated `permit` and `deny` lines.

Outlier Detection for Network Configurations: El-Arini and Killourhy [12] use a form of Bayesian inference to identify outlier configuration lines and demonstrate that the approach can find "lone commands," lines that only appear once in the given set of configurations. Le *et al.* [28] employ a data-mining algorithm to infer association rules; configurations that violate the rules are deemed outliers. These approaches identify misconfigurations in settings of interface definition and BGP sessions, including BGP route policies.

However, these approaches share two key limitations. First, they can only find outliers based on *exact equivalence*. Specifically, neither the approach of El-Arini and Killourhy nor the inferred association rules of Le *et al.* can infer configuration *parameters*, which is necessary to account for values that differ in expected ways across devices. Second, neither approach takes into account the importance of line reorderings within a configuration segment. Our structured generalization algorithm overcomes both limitations: bipartite matching of lines induces a natural form of parameterization, and sequence alignment of blocks enforces ordering constraints.

Mining Configuration Policies: Benson *et al.* show how to infer reachability policies for a network data plane [9]. Recently, Birkner *et al.* show how to infer similar policies that also take into account the control plane, ensuring that policies hold even in the presence of failures [10]. These *semantic* policies, which pertain to the end-to-end behavior of the network, are orthogonal and complementary to SELFSTARTER's *structural* policies for network configurations.

Benson *et al.* also introduce metrics and techniques to gauge the *complexity* of configurations [8]. Closest to our work is their technique to infer network *roles*. Their algorithm replaces field values with dummy entries — for example, IP addresses with the string "IPADDRESS" — and then employs an off-the-shelf clone detection tool [22] to find similar configurations. Our structured generalization is similar in spirit but provides several refinements necessary for template inference, including fine-grained support for parameterization and reordering. However, their work is complementary to ours, as we require the user to provide network role information.

Diagnosing Misconfigurations: Another line of work focuses on diagnosing the cause of misconfigurations. For example, NetPrints [5] does this through a form of decision tree learning, and PeerPressure [38] does this through a statistical analysis. Unlike SELFSTARTER, these tools start from a set of known or suspected misconfigurations, which the user must supply. Further, these tools diagnose misconfigurations in terms of a set of simple configuration features, while SELFSTARTER leverages the full structure of the configuration segments through template inference.

Automatic Differencing: Many algorithms exist for "diffing," for example, text comparison [32], clone detection [21, 22], and sequence alignment [36]. Our *structured generalization* algorithm combines some of these techniques in a novel manner, based on our domain requirements. We employ bipartite matching at the line level to support permutation and parameterization, but we introduce the *block* abstraction and perform sequence alignment on blocks to restrict certain reorderings while admitting insertions and deletions.

9 Conclusion

We presented an approach to identify misconfigurations in complex configuration segments, such as ACLs and route policies, without a specification. Such segments are typically intended to be similar across nodes playing the same *role*, yet they often have many intentional differences. We address this challenge by automatically inferring *templates*, modeling the (likely) intentional differences as variations within a template and the (likely) erroneous differences as variations across templates. Our *structured generalization* algorithm employs a novel two-level matching technique to allow controlled forms of parameterization and reordering within templates. To our knowledge this is the first approach to automatic template inference for network configuration segments.

Unlike the majority of work in network verification, which reasons about the *semantics* of networks, SELFSTARTER's analysis instead reasons about the *structure* of their configurations. While SELFSTARTER by design cannot understand packet behavior, it makes up for this lack by providing concise, actionable feedback directly in terms of the configurations. As a result, it has helped operators find and fix previously unknown network misconfigurations in three very different types of networks: datacenter, WAN, and campus.

Acknowledgments

Thanks to the NSDI reviewers and our shepherd Laurent Vanbever for their insightful comments, to the anonymous network operators for their feedback on SELFSTARTER's results, and to Shuang Zhang for help with SELFSTARTER's output visualization. This work was partially supported by NSF grants CNS-1704336 and CNS-1901510.

References

[1] Cisco blog | bgpmon.
https://bgpmon.net/blog/.

[2] Hyperscale cloud reliability and the art of organic collaboration.
https://www.microsoft.com/en-us/research/blog/hyperscale-cloud-reliability-and-the-art-of-organic-collaboration/.

[3] Intent-based networking in the data center: Cisco vs. juniper.
https://www.datacenterknowledge.com/networks/intent-based-networking-data-center-cisco-vs-juniper.

[4] Intent-based networking startups.
https://www.datacenterknowledge.com/networks/4-intent-based-networking-startups-innovating-disrupt-data-center-network.

[5] Bhavish Agarwal, Ranjita Bhagwan, Tathagata Das, Siddharth Eswaran, Venkata N. Padmanabhan, and Geoffrey M. Voelker. Netprints: Diagnosing home network misconfigurations using shared knowledge. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2009, April 22-24, 2009, Boston, MA, USA*, pages 349–364, 2009.

[6] Ehab Al-Shaer and Saeed Al-Haj. Flowchecker: Configuration analysis and verification of federated openflow infrastructures. In *Proceedings of the 3rd ACM Workshop on Assurable and Usable Security Configuration*, SafeConfig '10, pages 37–44, New York, NY, USA, 2010. ACM.

[7] 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, pages 155–168, New York, NY, USA, 2017. ACM.

[8] 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.

[9] 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.

[10] Rüdiger Birkner, Dana Drachsler-Cohen, Laurent Vanbever, and Martin Vechev. Config2spec: Mining network specifications from network configurations. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, 2020.

[11] Nikolaj Bjørner, Garvit Juniwal, Ratul Mahajan, Sanjit A. Seshia, and George Varghese. ddnf: An efficient data structure for header spaces. In Roderick Bloem and Eli Arbel, editors, *Hardware and Software: Verification and Testing - 12th International Haifa Verification Conference, HVC 2016, Haifa, Israel, November 14-17, 2016, Proceedings*, volume 10028 of *Lecture Notes in Computer Science*, pages 49–64, 2016.

[12] Khalid El-arini. Bayesian detection of router configuration anomalies. In *In Sigcomm Workshop on Mining Network Data*, pages 221–222. ACM, 2005.

[13] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, SOSP '01, pages 57–72, New York, NY, USA, 2001. ACM.

[14] 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 16)*, pages 217–232, Savannah, GA, 2016. USENIX Association.

[15] Nick Feamster and Hari Balakrishnan. Detecting bgp configuration faults with static analysis. In *Proceedings of the 2Nd Conference on Symposium on Networked Systems Design & Implementation - Volume 2*, NSDI'05, pages 43–56, Berkeley, CA, USA, 2005. USENIX Association.

[16] 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, 2015. USENIX Association.

[17] Fortune. American airlines network outage delays flights nationwide.
http://fortune.com/2018/07/29/american-airlines-network-outage/.

[18] 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.

[19] Phillipa Gill, Navendu Jain, and Nachiappan Nagappan. Understanding network failures in data centers: Measurement, analysis, and implications. In *Proceedings of the ACM SIGCOMM 2011 Conference*, SIGCOMM '11, pages 350–361, New York, NY, USA, 2011. ACM.

[20] 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. Validating datacenters at scale. In *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM '19, pages 200–213, New York, NY, USA, 2019. ACM.

[21] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pages 96–105, Washington, DC, USA, 2007. IEEE Computer Society.

[22] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. Ccfinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Trans. Softw. Eng.*, 28(7):654–670, July 2002.

[23] 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.

[24] 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.

[25] H. W. Kuhn. Variants of the hungarian method for assignment problems. *Naval Research Logistics Quarterly*, 3(4):253–258, 1956.

[26] H. W. Kuhn and Bryn Yaw. The hungarian method for the assignment problem. *Naval Res. Logist. Quart*, pages 83–97, 1955.

[27] P. Lapukhov, A. Premji, and J. Mitchell. Use of BGP for routing in large-scale data centers. Internet draft, 2015.

[28] Franck Le, Sihyung Lee, Tina Wong, Hyong Kim, and Darrell Newcomb. Detecting network-wide and router-specific misconfigurations through data mining. *Networking, IEEE/ACM Transactions on*, 17:66 – 79, 03 2009.

[29] 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, 2015. USENIX Association.

[30] Ratul Mahajan, David Wetherall, and Tom Anderson. Understanding bgp misconfiguration. In *Proceedings of the 2002 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '02, pages 3–16, New York, NY, USA, 2002. ACM.

[31] Haohui Mai, Ahmed Khurshid, Rachit Agarwal, Matthew Caesar, P. Brighten Godfrey, and Samuel Talmadge King. Debugging the data plane with anteater. In *Proceedings of the ACM SIGCOMM 2011 Conference*, SIGCOMM '11, pages 290–301, New York, NY, USA, 2011. ACM.

[32] Meld. Compare files, directories and working copies. https://git.gnome.org/browse/meld/tag/?h=release-0_5_1.

[33] James Munkres. Algorithms for the assignment and transportation problems, 1957.

[34] GD Plotkin. A note on inductive generalization, machine intelligence , editors b. *Meltzer, DI lichine, University PresB, Edinburgh*, Vol. 5:153–163, 1970.

[35] The Register. How four rotten packets broke centurylink's network for 37 hours, knackering 911 calls, voip, broadband. https://www.theregister.co.uk/2019/08/20/centurylink_outage_report_fcc/.

[36] T.F. Smith and M.S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147(1):195 – 197, 1981.

[37] Ars Technica. Google goes down after major bgp mishap routes traffic through china. https://arstechnica.com/information-technology/2018/11/major-bgp-mishap-takes-down-google-as-traffic-improperly-travels-to-china/.

[38] Helen J. Wang, John C. Platt, Yu Chen, Ruyun Zhang, and Yi-Min Wang. Automatic misconfiguration troubleshooting with peerpressure. In *6th Symposium on*

Operating System Design and Implementation (OSDI 2004), San Francisco, California, USA, December 6-8, 2004, pages 245–258, 2004.

[39] Hongkun Yang and Simon S. Lam. Real-time verification of network properties using atomic predicates. *IEEE/ACM Trans. Netw.*, 24(2):887–900, April 2016.