Poster: Identifying Syntactic Motifs and Errors in Router Configurations Using Graphs

Sara Alam
Colgate University
salam@colgate.edu

Devon Lee
Colgate University
dslee@colgate.edu

Aaron Gember-Jacobson
Colgate University
agemberjacobson@colgate.edu

***Abstract*—Router configurations are complex, so misconfigurations are common and hard to pinpoint. Existing configuration verifiers have several drawbacks. Consequently, we design a three stage heuristic to represent a wide range of components and relationships from a network's raw configurations as a graph, and we infer which of the aforementioned components refer to each other often by finding cycles in the graph. Deviations from frequently occurring cycles are considered misconfigurations.**

I. Introduction

Router configurations are notoriously complex. For example, routers in enterprise campus networks are often configured with multiple interfaces, routing protocols, access control list (ACLs), virtual local area networks (VLANs), and more [1]. Furthermore, configurations often contain numerous references between these components: e.g., interfaces participate in specific VLANs, VLANs apply specific ACLs, etc. [2]. Consequently, misconfigurations are common and hard to pinpoint.

Fortunately, researchers have developed two types of tools to detect router misconfigurations. Policy-checking tools (e.g., [3]–[6]) build models which encode a network's configurations and route computation algorithms and include model-specific analysis algorithms to check if predefined forwarding requirements are met. However, it is difficult to fully and accurately model the semantics of a network's configurations and route computation algorithms; even production-quality policy-checking tools contain errors [7]. Furthermore, policy-checking tools require a network's forwarding requirements to be specified, but requirements are rarely explicitly documented and tools for inferring requirements from configurations [8] may generate incomplete or inaccurate requirements [9].

Consistency-checking tools (e.g., [10]–[12]) check for deviations from recurring syntactic motifs to identify misconfigurations. By analyzing configurations' syntax instead of their semantics, consistency-checking tools eliminate the complex tasks of modeling the network's behavior and specifying a network's forwarding policies. However, existing consistency-checking tools focus on a narrow range of configuration components and relationships–e.g. only ACLs [12] or only BGP-related components [10]–causing these tools to overlook other syntactic motifs and corresponding misconfigurations.

Our goal is to design a consistency-checking tool which efficiently considers the full range of components and relationships present in the configurations of real networks. We make

```
hostname: case
interfaces: {
  GE1: {
    allowed-vlans: [100, 200],
    description: "Student workspace" }
  GE2: {
    allowed-vlans: [100, 200],
    description: "Student lounge" }
},
acls: {
  a: [
    remark "Student wireless",
    permit 10.0.20.0/24 ]
}
vlans: {
  100: {
    inbound-acl: a },
  200: {
    inbound-acl: a }
}
```

Fig. 1. Example configuration file

two significant contributions toward this goal. First, we design a three stage heuristic to infer a wide range of components and relationships from a network's raw configurations. We represent these components and their relationships as a graph, G. Second, we design a technique for efficiently analyzing G to identify misconfigurations. In particular, we infer which of the aforementioned components refer to each other often by finding cycles in G. Deviations from frequently occurring cycles are considered misconfigurations.

In the rest of this abstract, we describe our approach in more detail (Section II) and present our preliminary findings for a small university campus network (Section III).

II. Our Approach

A. Building G

We design a three-step heuristic to convert raw configurations into an easily-analyzable structure. First, we extract configuration components (e.g., interfaces, ACLs, and VLANs) from a JSON representation of a networks' configurations by inferring whether the "key" at a particular level of indentation is a "type" of component or the "name" of a specific component. We leverage the fact that components of the same type typically have overlapping attributes and components of different types have distinct attributes. For example, in the configuration in Figure 1, interfaces and VLANs are identified as separate types of components because of their

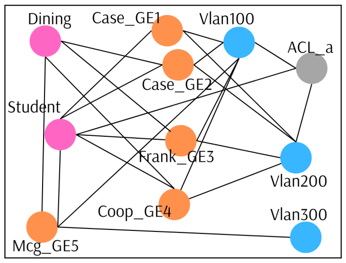

Fig. 2. An example graph with nodes of type interface (orange), VLAN (blue), ACL (gray) and keyword (pink)

different attributes—`allowed-vlans` and `description` versus `inbound-acl`—and `GE1` and `GE2` are identified as names of interfaces. Unlike prior consistency-checking tools, we explicitly include components with no semantic impact (e.g., interface descriptions), because they provide useful insights into a network's configurations (e.g., interfaces with the same role may have the same keyword in their description).

Second, we check which pairs of components refer to one another at least once by looking for the names of one component within the details of another. For example, in Figure 1, interface GE1 refers to VLAN 100.

Third, we create a graph G, which contains a node for every name of every type of component extracted and an edge between every pair of names which refer to each other. Our approach is more holistic compared to existing policy- and consistency-checking tools, because we include *all* the components mentioned in the configurations files.

B. Analyzing G

A multi-component relationship can be viewed as a series of edges in G that ties a set of nodes together. In other words, common ways that components are grouped together in a given network can be characterized as cycles in G. Cycles that appear most often are identified as network specific patterns. Instances of these patterns where the cycle-completing edge is missing are flagged as possible misconfigurations.

To find cycles, we pick a specific "anchor" type and cycle length (k). We use Breadth First Search to enumerate all k-length paths starting from each instance of the anchor type. The paths which lead back to the starting nodes with one additional edge are full cycles; the rest are partial cycles. We group paths which have the same "signature"—i.e., sequence of node types and/or names—and compute a "confidence" ratio: Num full cycles / (Num partial cycles + Num full cycles).

For example, starting from interface (orange) nodes in Figure 2 there are ten paths with the signature "interface $\rightarrow$ Student $\rightarrow$ ACL_a $\rightarrow$ VLAN". One of the paths (Mcg_GE5 $\rightarrow$ Student $\rightarrow$ ACL_a $\rightarrow$ Vlan200) is a partial cycle and the rest are full cycles, so the confidence is 90%. If the confidence is 100%, the cycle is a recurring motif, but there are no associated misconfigurations. If the confidence is low or the number of partial cycles is high, then the deviations highlighted occur often, and it is unlikely to be a misconfiguration. We have observed that cycles with confidence between 90% and 100% exclusive, with the number of partial cycles below 100, are the best at highlighting misconfigurations without flagging correct configuration components.

An important aspect of cycle-finding is that a mix of names and types must be used to define each signature. Consider all-name and all-type signatures as two possible extremes. Every node in an all-name signature is unique, so this path will only appear once and not form a recurring pattern: e.g., Case_GE1 $\rightarrow$ Vlan100 $\rightarrow$ Frank_GE3 $\rightarrow$ Student $\rightarrow$ Case_GE1 in Figure 2. In contrast, all-type paths (e.g., interface $\rightarrow$ keyword $\rightarrow$ acl $\rightarrow$ vlan) are often too general to frequently form full cycles and capture meaningful patterns. Thus, we consider signatures with a combination of names and types.

III. Preliminary Results & Future Work

We have implemented a prototype of our approach in Python and evaluated it using configurations from a small university campus network. In particular, we looked for 3- and 4-node cycles with interface nodes as anchors. Most of the 3-node cycles in this network did not uncover any misconfigurations. Our tool found 128 4-node cycles with confidence between 90% and 100% exclusive. Network engineers have confirmed that all of the anomalies found are either actual misconfigurations or exceptions which are beneficial to track. The time taken to discover paths grows linearly with the number of anchor nodes and superlinearly with the number of relationships between nodes. However, path discovery can be easily parallelized, allowing our approach to scale to larger networks.

In the future, we plan to generalize our analysis algorithm to: i) Identify a suitable maximum cycle length (n) for a network; ii) Find types that give the most relevant cycles when used as anchors; iii) Enumerate and evaluate all possible k-node cycles using the set of types for all k such that $3 \leq k \leq n$.

References

[1] D. A. Maltz, G. G. Xie, J. Zhan, H. Zhang, G. Hjálmtýsson, and A. G. Greenberg, "Routing design in operational networks: a look from the inside," in *ACM SIGCOMM*, 2004.
[2] T. Benson, A. Akella, and D. Maltz, "Unraveling the complexity of network management," in *USNEIX NSDI*, 2009.
[3] R. Beckett, A. Gupta, R. Mahajan, and D. Walker, "A general approach to network configuration verification," in *ACM SIGCOMM*, 2017.
[4] A. Abhashkumar, A. Gember-Jacobson, and A. Akella, "Tiramisu: Fast multilayer network verification," in *USENIX NSDI*, 2020.
[5] S. Prabhu, K.-Y. Chou, A. Kheradmand, B. Godfrey, and M. Caesar, "Plankton: Scalable network configuration verification through model checking," in *USENIX NSDI*, 2019.
[6] P. Zhang, D. Wang, and A. Gember-Jacobson, "Symbolic router execution," in *ACM SIGCOMM*, 2022.
[7] F. Ye, D. Yu, E. Zhai, H. H. Liu, B. Tian, Q. Ye, C. Wang, X. Wu, T. Guo, C. Jin *et al.*, "Accuracy, scalability, coverage: A practical configuration verifier on a global WAN," in *ACM SIGCOMM*, 2020.
[8] R. Birkner, D. Drachsler-Cohen, L. Vanbever, and M. Vechev, "Config2Spec: Mining network specifications from network configurations," in *USENIX NSDI*, 2020.
[9] A. Kheradmand, "Automatic inference of high-level network intents by mining forwarding patterns," in *ACM SOSR*, 2020.
[10] N. Feamster and H. Balakrishnan, "Detecting BGP configuration faults with static analysis," in *USENIX NSDI*, 2005.
[11] F. Le, S. Lee, T. Wong, H. S. Kim, and D. Newcomb, "Detecting network-wide and router-specific misconfigurations through data mining," *IEEE/ACM Trans. Netw.*, vol. 17, no. 1, pp. 66–79, 2009.
[12] S. K. R. K., A. Tang, R. Beckett, K. Jayaraman, T. D. Millstein, Y. Tamir, and G. Varghese, "Finding network misconfigurations by automatic template inference," in *USENIX NSDI*, 2020.