Maverick: Detecting Network Configuration and Control Plane Bugs Through Structural Outlierness

Vasudevan Nagendra
Plume Design Inc.
Palo Alto, USA
vnagendra@plume.com

Abhishek Pokala
Stony Brook University
Stony Brook, USA
apokala@cs.stonybrook.edu

Arani Bhattacharya
IIIT Delhi
New Delhi, India
arani@iiitd.ac.in

Samir Das
Stony Brook University
Stony Brook, USA
samir@cs.stonybrook.edu

***Abstract*—Proactive detection of network configuration bugs is important to ensure the proper functioning of networks and reducing the issues associated with network outages. In this research, we propose to build a control plane verification tool `MAVERICK` that detects the bugs in the network device configurations by effectively leveraging *structural deviation* i.e., *outliers* in the network configurations. `MAVERICK` automatically infers *signatures* from control plane configurations (e.g., Access Control Lists (ACL), route-maps, route-policies, and so on) and allows administrators to automatically detect bugs in the network configurations with minimal human intervention. The outliers calculated using *signature-based outlier detection* mechanism are further characterized for its severity and ranked or re-prioritized according to their criticality. We consider a wide set of heuristics and domain expertise factors for effectively reducing the false positives. Our evaluation on four medium to large-scale enterprise networks shows that `MAVERICK` can automatically detect the bugs present in the network with ≈86.4% accuracy. Furthermore, with minimal administrator inputs i.e., with a few minutes of signature re-tuning, `MAVERICK` allows the administrators to effectively detect ≈92 – 100% of the bugs present in the network, thereby ranking down less severe bugs and removing false positives.**

Keywords*— *Network; Control Plane; Verification; Outliers; Machine Learning; Anomaly Detection; Bugs; Severity.

I. INTRODUCTION

Network downtime for an enterprise network costs an average of USD $140K – $500K per hour, for which the human error acts as the key contributing factor [1][2]. The fundamental goal of network management and downtime mitigation is *proactive detection* of the control plane and network configuration bugs, and ability to quickly troubleshoot the errors that occurred due to human errors and misconfigurations. Today network administrators either rely on custom *home-made* scripts 'or' model checking-based verification tools for analyzing the network configurations to detect specific types of bugs in the network (e.g., reachability analysis, routing issues, failure impact analysis, and so on) [3][4][5][6]. Such tools provide limited bug detection capability i.e., does not provide comprehensive coverage about the list of bugs present in the network configurations. Therefore, a generic control plane bug detection mechanism that proactively detects a comprehensive list of bugs present in the network with minimal administrator's intervention is key for protecting the networks from downtime and vulnerabilities.

Traditionally, bug detection can be efficiently achieved by defining unique signatures to each of the network properties and matching each of the configuration instances with respective signatures. For example, an ACL that allows web traffic from LAN network to Internet needs to be specified on to a group of network devices along the path of the traffic until the traffic reaches the border gateway of the enterprise network. Therefore, multiple devices should have either same or similar ACL and deviation from the actual ACL definition would be considered a bug. As a similar example, route maps are used for defining the set of route entries that are required to be redistributed to target routing process, requiring the route maps to be specified on to multiple routers.

Manually identifying such signatures (or specifications) in a dynamically changing network infrastructures and effectively using such comprehensive list of signatures for detecting bugs is a daunting task. But, not providing *signatures (i.e., about what specifically needs to be looked for in the network configurations)* results in bugs and errors that either go undetected (with false negatives) or results in false positives that plague the soundness of the bug detection tool. In our observation, there are legions of bugs that remain undetected even with networks "vetted" by verification tools, because of a lack of capability that allows the signature to be specified and used for bug detection.

Therefore, the current network verification tools falls short along following key dimensions: (i) proactively detecting control plane bugs (e.g., human errors and configuration mistakes) without (or with minimal) administrator's intervention, (ii) ability to effectively incorporate domain expertise in fine-tuning the bug detection, (iii) automatically inferring policies or signatures from the network configurations that allows administrators and tools to effectively detect configuration bugs, while providing comprehensive bug detection coverage, (iv) generalize findings, i.e., signatures or policies inferred from one network and apply it to other networks or organizations, and (v) finally, surfacing the bugs that are critical allowing administrators to channelize their energy in addressing critical bugs rather than wasting time on false negatives.

To address the above challenges, we propose `MAVERICK`, an agile network verification tool that exploits structural deviations (i.e., *Outlierness*) among the network configurations for detecting the bugs. *Outlierness* is the deviation of the network configurations from its general population or most popular values. The key enabler of `MAVERICK` is its ability to automatically infer signatures from the network configurations, which are used for efficiently detecting bugs present in the network, without false negatives. `MAVERICK` also incorporates inputs from the administrators allowing the tool to fine-tune its detection precision. In addition, `MAVERICK` also proposes the need for generalization by which the signatures that are developed for an network can be used with other networks.

We improve the accuracy of our bug detection mechanism and efficiently re-prioritize the bugs to surface them to administrators on the basis of their severity. We calculate severity of the bugs using following key metrics, such as feature importance (i.e., network structural properties such as ACLs, route-maps, IPSec tunnel configurations and so on), feature dependency, the locality of the configuration on specific node, outlierness score from the similarity with signatures, and customized page ranking used for ranking bugs. These metrics allow `MAVERICK` to effectively prioritize bugs on the basis of their severity, pushing false positives or less critical bugs to the bottom

 ISBN: 978-1-61208-919-5

of the list. We prove the efficacy of MAVERICK by showing that it provides a mean precision of 86.4% without administrator input, and 92 – 100% using a few minutes of administrator input on a four medium to large-scale enterprise networks.

In summary, our paper makes following key contributions:

- We provide background and illustrate the limitations of existing techniques and motivate the need for signature-based bug detection mechanisms based on outliers (§II).
- We highlight the techniques we used to automatically infer the signatures for various properties of network configurations using their *structural outlierness* for detecting the bugs. We then discuss about simple severity and ranking mechanism that we devised to reprioritize the bugs and reduce the false positives (§III)
- We discuss about the high level system design and key building blocks of MAVERICK (§IV).
- We evaluate the efficacy of MAVERICK with four different medium – large scale campus and enterprise networks with ≈220 – 450 network nodes (e.g., routers, firewalls, switches, proxies, and gateway nodes) (§V).

II. Background & Motivation

Today, majority of the network administrators still rely on plain-text configuration templates, command-line utilities, and wide variety of vendor-supplied programming specifications or user-interfaces for programming their networks [7][8][9] [10]. This results in administrators unintentionally introducing bugs in the network configurations resulting in network outages or leaving the network vulnerable to attacks [11][12].

We understand that for programming the network and creating policies requires same set of rules to be specified on wide range of devices that are present with in a network. Consider for example, an ACL that is specified to allow TCP traffic that is destined to WAN network 100.100.100.0/24 on port 1400 requires a group of same ACLs to be specified on multiple routers or firewalls along multiple paths in which the traffic traverses. Similarly, route-map entries, NAT rules, and route-filters specific to this ACL are also required to be specified on these routers along the paths in which the traffic traverses. In general, administrators either use sample templates or use CLI to configure multiple routers, which may result in human introduced errors.

Router_1 (Key-value Properties):

```
{
DNS Servers: ['4.4.4.4']
NTP Servers: ['0.pool.ntp.org' ,
'1.pool.ntp.org']
TACACS Servers: ['10.10.10.15']
Logging Servers: ['10.10.10.22']
}
```

(a) Network Server Properties.

Router_1 (Named Structure Property):

```
{
'action': 'PERMIT',
'matchCondition=dstIps=ipWildcard':
[('100.100.100.0/0', 24)],
'matchCondition=ipProtocols': ['6']
'matchCondition=tptDstPort': [('1400')]
}
```

(b) IP ACL.

Figure 1: Illustrating *key-value* properties (e.g., Network Server values) and *named-structure* properties (e.g., IP ACL) in network configurations.

We broadly classify overall network configurations into two property classes (Figure 1): (*i*) *Key-value properties*, and (*ii*) *Named-structure properties*. As illustrated in Figure 1a, *key-value* property is a simple key:value/s pair that represents a discrete and independent network configuration (e.g., NTP Server configured for Router_1 in Figure 1a). While the *named-structure* properties are structures with multiple key:value pairs nested as a complex discrete entity required to configure the network.

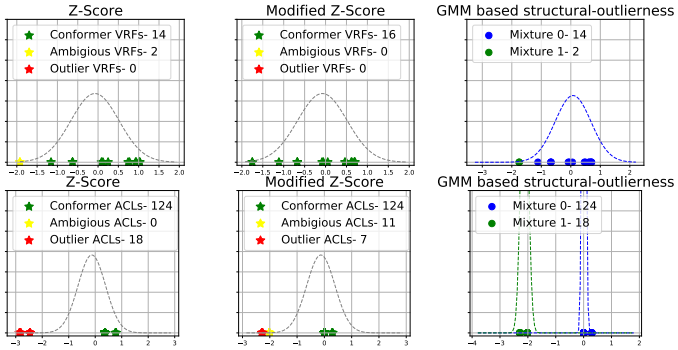

Figure 2: Bug detection using statistical approaches (z-score, modified z-score, GMM) for VRFs, ACLs of network (DS-1).

A. Problems with existing approaches

For effectively detecting the bugs present in the network configurations, existing approaches aim to supplement the manual effort of network administrators by flagging probable network configuration and data plane bugs [13][14], which broadly fall into two categories: (*a*) Statistical approach, and (*b*) logical or rule-based approach.

Statistical techniques. Statistical approaches such as z-score, modified z-score and Gaussian Mixture Model (GMM) aim to identify outliers in the configurations, and flag them as probable bugs [15][16][17][18] as illustrated in Figure 2. While we note that the outlying configurations have a much higher probability of being bugs, that in itself is not sufficient to detect real bugs and highlight its severity. The key disadvantages of such statistical approaches are as follows:

- *High mis-classification rate*: Since many of the configurations lie at the boundary of the threshold used to classify as bugs, a large number of either false positives (i.e., incorrectly flagged as bugs) or false negatives (i.e., incorrectly flagged as valid) are identified. Correctly identifying the actual bugs from these lists again requires a lot of manual effort on the part of the administrator.
- *Flagging intentional configuration changes*: Administrators might intentionally change configurations in specific ways to handle an uncommon use case. However, statistical techniques identify even such changes as configuration bugs.
- *Critical bugs vs false positives*: In general, not all network configuration bugs are equally critical. Some bugs require immediate attention from administrators, whereas other bugs can be fixed slowly. However, we lack mechanism to identify the bugs that are critical in nature.

Logical or rule-based techniques. This approach is to let users specify grammar rules, any violation of such grammar rules is flagged as a configuration bug [4][19][20][21]. However, this approach too suffers from a number of drawbacks:

- *Requirement of low-level vendor-specific rules*: We see different vendors using different syntax to specify network configurations which introduces an additional level of complexity in specifying these rules. It requires administrators to specify complex low-level grammar rules. Thus, this is usually a cumbersome and technically involved process, that is also prone to mistakes.
- *Lack of coverage*: Even for proficient administrators, it is challenging to anticipate all the types of invalid configurations and proactively fix them. Thus, many configuration bugs may pass through without getting identified.

Therefore, it is becoming increasingly difficult to proactively detect the network configuration bugs before deploying them on to the production networks. In Section III, we present the overview of MAVERICK system that addresses the challenges discussed here.

 ISBN: 978-1-61208-919-5

III. MAVERICK Overview

We present the overview of MAVERICK control plane network verification engine that is a tangible step toward addressing the limitations discussed above (§II-A). Figure 3 provides an overview of the MAVERICK system architecture, with the following two key capabilities: (i) *signature-based outlier detection* engine, and (ii) *severity & ranking* engine. These capabilities allow administrators to proactively detect control plane bugs and fine-tune them to reduce the false positives, while reducing their time vested in triaging critical bugs rather than spending time on false-positives.

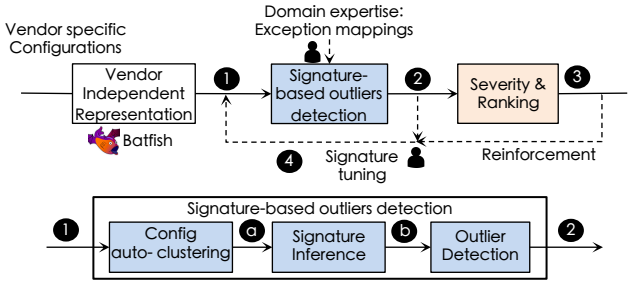

Figure 3: MAVERICK System Architecture.

Signature-based outlier detection. This module digests the network configurations provided in vendor-specific format and translates them to *vendor-independent* specification for detecting the outliers in the network configurations (❶). We leverage the specification mechanism used in batfish network verification tool [4]) for representing the configurations in vendor-independent format. We calculate the *structural outlierness* among the network configurations (i.e., represented in vendor-independent format) for effectively detecting bugs in the networks. We define, *structural outlierness* as the deviation of a network property (i.e., key-value property or named-structure) from its group or cluster of configurations (i.e., most popular entries of the cluster) that are programmed onto multiple nodes to achieve the same functionality (discussed in §II) (ⓐ).

For the derived most popular entries within a group or cluster, we apply domain expertise (i.e., captured as *exception mappings*) for automatically inferring the signatures (ⓑ). We supply such automatically inferred signatures to administrator for inspection and fine-tuning these signatures for detecting bugs in the network configurations (❷). Though administrator's intervention is optional in our case, we use *human-in-loop* for reducing false positives (❹). These signatures we inferred could be used to detect bugs in the network configurations before applying them to the network (*Signature-based outlier detection*). The capabilities discussed above are performed by following three key modules of *signature-based outlier detection* engine (see §IV): (a) *Config auto-clustering* module, (b) *signature-inference* engine, and (c) *Outlier detection* engine.

Severity & ranking. For the bugs that are detected from the *signature-based outlier detection* module, we apply the severity and ranking mechanism that we developed to re-prioritize the bugs for identifying their severity. Deriving severity of each bug helps to reduce the administrator's effort and time in handling false positives (❸). We use following three key metrics for calculating the severity for ranking the outliers (see §IV-B): (a) *Similarity and outlierness scores*, (b) Well connected-ness of nodes, (c) *Feature-dependency*.

Design goal. Our goal is to mitigate the problems in existing enterprise and campus networks by reducing the amount of effort involved in detecting bugs, automatically inferring signatures that acts as reference to verify the network configurations for its sanity and bugs, while increasing the network coverage (for detecting generic bugs). Unlike, existing techniques which requires network configurations to be manually grouped for building the templates [15], MAVERICK automatically clusters the configurations into separate groups for building the signatures. However, a key drawback of such signature inference is that it falsely flags configurations that network administrators have designed for customized use cases. To mitigate this problem, we allow administrators to re-tune inferred signatures. Since there can be multiple valid signatures, this also automatically allows more customized configurations.

We recognize that even with multiple signatures, it is possible to false classify multiple configurations as bugs. However, not all configuration bugs are equally important in a network. Based on the estimated severity score, we assign priority to each of the identified bugs and rank them accordingly. This allows the administrators to focus on the most important bugs, while letting the less important ones remain for longer time.

IV. High Level System Design

In this section, we discuss the network verification mechanism that we developed to address the limitations discussed in §II-A. As shown in Figure 3, MAVERICK supports following key functional components to address these limitations: (i) Signature-based outlier detection, and (ii) Severity and ranking mechanism.

A. Signature-based Outlier Detection

We use the specification language discussed in the Batfish [4] to translate the network configurations from vendor-specific languages (e.g., Cisco's IOS, Juniper's JunOS) to vendor-independent (VI) representation, which avoids the need for designing parsers for each of the vendor-specific language in MAVERICK. We extract network configurations i.e., *named structure properties* (e.g., ACL's, route-maps, route-policies), and *network server properties* (e.g., DNS server, NTP server, Authentication servers), and configurations on all the network devices and encode such categorical data into binary encoded format i.e., using Multi-label binarizer [22], which allows us to apply statistical and Machine Learning (ML) techniques on the network configuration data.

The key challenge in bug detection is the ability of administrator to craft the specification or signature that allows the tool to detect the bugs and errors. Therefore, automatically generating (i.e., inferring) the signatures is the key step towards effective detection of bugs in the network configurations. MAVERICK's signature-based outlier detection engine supports following three key capabilities for automatically detecting the bugs present in the network configurations represented in vendor independent format.

Configuration auto-clustering. As a first step, we run clustering on each of the named structures (such as ACLs, router-filters, route-maps) independently, to group them on the basis of their categories and properties. For example, a network with thousands of ACLs are clustered into group of tens or groups of hundred on the basis of their similarity i.e., for automatically inferring signatures from each of the ACL groups, which is required to compute its signature. As manually grouping thousands of ACLs into groups on the basis of their name or other properties is a challenging and tedious process, we use simple K-means a ML-based technique to cluster the named structures. The clustered named structures are then used for signature inference. To obtain the right value of K, we use Elbow [23], and Silhouette [24] methods to regress on different values of K to decide the optimum. We heuristically choose a lower limit of K (i.e., regressed from the above three techniques) equal from the number of unique set of names used to configure different named structures. Therefore, clustering reduces the number of signatures inferred, thereby reducing the amount of manual effort involved with administrator in verifying the signatures to re-tune them for increasing the precision of signature-based outlier detection.

Signature inference & generalization. The signature inference engine automatically infers and builds the signatures from the clustered

 ISBN: 978-1-61208-919-5

Algorithm 1 Signature Inference Algorithm.

```
F ← generateVI()
P ←getNamedStructProps(F)
P ←encode(P)
K ← elbow(P)
C ← clusters using K-Means of P
Let F(c, p) be the value-frequency pair ∀c ∈ C, p ∈ P.
Compute threshold T(c, p) from F(c, p), ∀c ∈ C, p ∈ P.
Let ϵ be the margin of uncertainty
for c ∈ C do
    for p ∈ P do
        for (k, v) ∈ F(c, p) do
            if v > T(c, p) + ϵ then
                Mark p as bug
            else if v > T(c, p) & v < T(c, p) + ϵ then
                Mark p as probable bug
            else
                Mark p as normal property
```

named structures. The signature inference engine composes all the named part of the cluster to frame a single signature. We use following grammar in our signature for effectively capturing and generalizing the signatures, which includes following operators: '*', '!' '=' '[]', '{}', 'OR', 'AND', ¡IP-Subnet¿ (i.e., IP specific to that subnet will be considered as legitimate in the signature).

As shown in Figure 4, the signature of a named structure includes set of key-value pairs (i.e., complex nested). The Key is property name and the values are array of tuples. The tuples captures one of the values of property and its weight, where as weight represents the frequency of occurrences or the density of the value for that property with in that cluster.

```
IP_ACL_1 (Signature): {
action: {PERMIT: 45},
matchCondition=Class: {temp1: 36, temp2: 1, temp3:8},
matchCondition=headerspace=ipProtocols: {TCP: 36, UDP: 9],
matchCondition=headerSpace=tcpFlagsMatchConditions: {True: 1},
...
srcPorts: [51102-51102: 37, 51102-51103: 5 51102-51104: 3]
}
```

Figure 4: Signature Inferred by MAVERICK for IP ACL with the popularity weights are shown above. Only part of the signature is shown for brevity.

Also, the ability of these techniques to effectively accommodate the domain expertise and inputs from administrators allows them to effectively detect bugs present in the network. The signature-mappings enforces constraints on the property's key:value pairs that are part of the signature. The signature-mapping which is provided as the domain knowledge from the administrator restricts the signature inference engine to treat specific key:value pairs differently. For example, the inference engine can discard any specific key and value associated with it from being part of the signature. For example, we do not want our bug detection engine to consider the configuration patch added by administrator to specific issue or corner as outliers. This allows us to *white-list*, create *exception*, or *black-list* specific keys to the signature inference engine about the way it should consider the respective key:value pairs.

Re-tuning outlier detection. Signatures auto-generated using ML-based techniques could be further fine-tuned by administrator by supplying the domain knowledge as *signature-mappings* or manual inspection. On the contrary, for simple server properties (e.g., DNS servers, TACACS server properties) the names used on different nodes are required to be same, which simplifies our task of grouping configurations for clustering to detect outliers. Hence, they could be simply grouped together for calculating the outliers.

To verify if a named structure is an outlier, we compare the properties of this named structure with the respective properties of the cluster signature. If all the properties in the named structure that is compared with the signature matches, then the named structure is considered as valid and bug otherwise. We also calculate their similarity scores S_i and outlier scores O_i, to determine the amount by which a named structure matches with the signature.

$$S_i = \frac{\sum_{i=1}^{n} W_i}{\sum_{j=1}^{s} W_j}, O_i = 1 - S_i, \forall i = 1, \ldots, n; \ \forall j = 1, \ldots, s, \quad (1)$$

where n is total number of properties in the signature, s is total number of signatures, and W_i represents the weight associated with each of the property in the signature.

B. Severity and Ranking

This list of outliers that is generated as outcome of the signature-based outliers engine contains the outlier definition, the named structure it belongs to, and it's outlier score. The outlier score is an indication of how strongly our engine believes a particular outlier to be a bug and its value is between 0 and 1. But an entry with a very high score could mean that it is a single separate configuration and does not belong to any signature. Our severity and ranking mechanism takes this into consideration for effectively calculating the severity. To rank these outliers, we devise different metrics and assign each outlier a metric score. Then, using a particular combination of these metric scores, we calculate the final score of each outlier and rank them based on this score. MAVERICK uses following three different metrics to calculate the severity and ranking of the outliers:

(i) *Similarity and outlierness scores* that we derived from the outcome of signature-based outlier engine is used as one factor in deciding the severity of the final bug outcome.

(ii) *Well-connectedness* of nodes: We use the page-rank algorithm to establish the importance of each node with the general idea being that a possible bug in a more important or well-connected node would be more severe than a bug that has fewer connections.

(iii) *Feature-Dependency Score*: This metric tells us the importance of the features that the named structure is a part of. The general idea is that the importance of named-structure is network-specific and therefore, dynamically evaluating these scores helps provide a much finer and network-specific bug severity analysis. Consider for example, when a ACL rule marked as outlier will results in impacting the NAT rules, route-filters and VRFs associated with it. Hence, outliers in features that has higher dependency with other features will result in high severe bugs. The final outcome of the severity and ranking module results in generating bugs that result in lesser in FPs and FNs (see TABLE I) and effectively ranked according to its severity (Figure 5).

Finally, the *human-in-the-loop* correlation score helps re-tune the signature and reduces false-positives. Once the network administrator flags a certain outlier as a bug or a FP, all the corresponding outliers in the population (i.e., cluster in our case) show an increase or a decrease in their severity score respectively. This metric allows the administrator to manually inspect numerous bugs of a specific type from a very large network with relative ease.

V. Prototype Evaluation

Dataset. We evaluate the performance of MAVERICK over a total of four networks using their network configuration. Of the four networks, three are of medium size network of 157, 132 and 221 nodes (e.g., switches, routers, firewalls, etc.,) and large network of 454 nodes. Medium networks has around 5000 – 10000 properties, while large scale network has around 60000 properties. The properties consist of ACLs, Route Filters, VRFs and Routing Policies with

TABLE I: FINAL RANKED BUG OUTCOME MAVERICK TOOL IN ACCORDANCE WITH ITS SEVERITY.

Outlier	Signature Definition	Conformer Nodes	Outlier Definition	Outlier Nodes	Outlier Properties	Outlierness Value	Severity Score
outlier:Route_ Filter_List_0	{'action': [['PERMIT', 16]], 'ipWildcard':[['100.100.100.0/23', '*', 9], ... ['25-25', '*', 10]]}	['rt1-dc1', 'rt2-dc1'. .., rt91-dc1]	{'action': 'PERMIT', 'ipWildcard': '100.100.0.0/16', 'lengthRange': '16-20'}	['rt19-dc1', 'rt28-dc1']	[['lengthRange', '16-20']]	0.978	1.177

ACLs predominant in the large network, whereas RouteFilters are predominant in the medium sized networks.

Performance. We first compare the performance of MAVERICK in terms of precision and recall for comparison with baseline techniques for medium scale enterprise network dataset (DS-3) (see TABLE II)

The baseline techniques include Z-score, modified Z-score, GMM, and MAVERICK using signature-based outliers. We make two major observations: (i) We note that MAVERICK's outlier detection performs better than Z-score, modified Z-score and GMM in terms of both precision and recall. However, the precision is still only around 0.86, which has further scope of improvement. This is primarily due to presence of false positives, as a large number of outliers are detected using the inferred signatures. (ii) Manual retuning of signatures can then further increase the precision to 0.92, thus increasing by additional 7 – 8% compared to just outlier-based detection. Careful retuning of ≈97 clusters/signatures detected by MAVERICK for DS-3 required less than 2 hours for manual inspection. This shows that manual retuning of signatures can further improve the precision.

TABLE II: EFFICACY OF MAVERICK FOR MEDIUM SCALE ENTERPRISE NETWORK DATASET (DS-3).

Approach	TP	FP	FN	Precision	Recall
Z-score	392	1031	240	0.275	0.620
Modified Z-score	417	692	132	0.386	0.760
GMM	298	608	220	0.329	0.575
Maverick (Outliers)	472	74	32	0.864	0.937
Maverick (Retuning)	498	32	8	0.92	0.984

Severity score. We now look at how severity score can change the sequence of bugs shown to administrators (Figure 5). We plot the bugs reported in the sequence of their outlier score, along with their severity scores for one of the medium-sized network. We note that the sequence of bugs shown to the administrators changes considerably, with P16 rising up to the most severe rank, followed by P15 and P13. On the other hand, P6 reduces to the least severe rank, followed by P5. This shows that using severity score alters the sequence of bugs shown to the users, and can lead to less important bugs being given less priority even if they have high outlier scores.

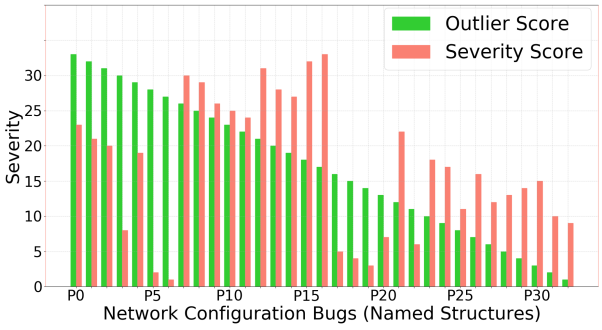

Figure 5: Bugs discovered by MAVERICK with and without severity applied.

Correlation of outliers with real network issues. To further observe the type of bugs MAVERICK discovers, we utilize a sankey diagram to show how outliers in different properties correspond to different types of bugs (Figure 6) in one of the medium-sized network (DS-3). Correlation of bugs discovered by MAVERICK with real-world network problems. The left layer corresponds to the type of property in which outlier is detected. The middle layer corresponds to the type of signature that is violated. The right layer is the type of real-world network problem. A higher thickness of flow denotes a higher number of bugs corresponding to a specific signature in the middle layer of vertices and then to types of network problems in the last layer. We observe that most of the bugs arise due to problems in IP access lists, followed by routing policy, route filter list and VNF's. We also observe that the most common type of network problem is undefined references, but each type of outlier roughly has equal probability of leading to an undefined reference.

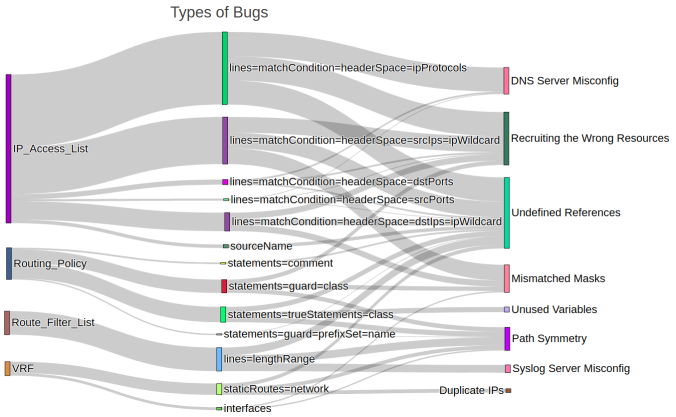

Figure 6: Correlation of bugs discovered by MAVERICK.

VI. CONCLUSION

This paper presented a novel signature inference framework for detecting the control plane bugs based on structural deviations (i.e., outliers or bugs), while their severity is estimated and bugs are ranked accordingly. The key strength of this work lies in its ability to automatically infer the signatures from raw network configurations without much administrator's intervention and generalize these inferred signatures for transportability. We combine disparate metrics to rank the severity of the detected outliers. We evaluated our approach using four different datasets of campus networks and achieved high bug detection of up to 92% with supply of domain expertise in the form of signature-mappings. While our approach was simple, with inferred signatures we were able to discover numerous bugs, including those that would be impossible to discover with existing network validation tools.

We made our tool and overall framework that supports wide range of statistical and ML algorithms along with signature-based outlier analysis tool as open source to stimulate additional research specifically in enhancing the network verification, and control and data plane bug detection mechanism [25].

ACKNOWLEDGEMENTS

This work was done by Vasudevan Nagendra, Abhishek Pokala, and Arani Bhattacharya when at WINGS Lab, Computer Science department, Stony Brook University. This work was partially supported by NSF grant CNS-1642965.

REFERENCES

[1] The Cost of Downtime. https://blogs.gartner.com/andrew-lerner/2014/07/16/the-cost-of-downtime/, July 2014.

[2] The Ugly Truth about Downtime Costs and How to Calculate Your Own. https://www.itondemand.com/2018/05/29/costs-of-downtime/, May 2018.

[3] A. Gember-Jacobson, R. Viswanathan, A. Akella and R. Mahajan. Fast control plane analysis using an abstract representation. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 300–313. ACM, 2016.

[4] A. Fogel, S. Fung, L. Pedrosa, M. Sullivan, R. Govindan, R. Mahajan, and T. Millstein. A general approach to network configuration analysis. In *12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15)*, pages 469–483, 2015.

[5] A. Panda, K. Argyraki, and M. Sagiv, M. Schapira, and S. Shenker. New directions for network verification. In *1st Summit on Advances in Programming Languages (SNAPL 2015)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.

[6] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and B. Godfrey. Veriflow: Verifying network-wide invariants in real time. In *Presented as part of the 10th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 13)*, pages 15–27, 2013.

[7] Use templates to define a common device configuration. *Product Documentation*, 2017.

[8] Managing Multiple Networks with Configuration Templates. *Product Documentation*, 2018.

[9] Google Cloud Networking Incident #19009. *Google Cloud Networking Incidents*, 2019.

[10] 451 Research: Enterprise Network Automation Gets Competitive. *Technical Report*, 2020.

[11] A. Bednarz. Top reasons for network downtime: Network outages linked to human error, incompatible changes, greater complexity. *Technical Report*, 2018.

[12] A. Patrizio. The biggest risk to uptime? Your staff: Human error is the chief cause of downtime, a new study finds. Imagine that. *Technical Report*, 2019.

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

[14] Y. Li, X. Yin, Z. Wang, J. Yao, X. Shi, J. Wu, H. Zhang and Q. Wang. A survey on network verification and testing with formal methods: Approaches and challenges. *IEEE Communications Surveys Tutorials*, 21(1):940–969, 2019.

[15] S. Kakarla, T. Alan, R. Beckett, K. Jayaraman, T. Millstein, Y. Tamir and G. 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.

[16] C. Christian, S. Vaton, and M. Pagano. A new statistical approach to network anomaly detection. In *2008 International Symposium on Performance Evaluation of Computer and Telecommunication Systems)*, pages 441 – 447, 2008.

[17] booktitle=International Static Analysis Symposium T. Kremenek, and D. Engler. Z-ranking: Using statistical analysis to counter the impact of static analysis approximations. pages 295–315. Springer, 2003.

[18] T. Kremenek, K. Ashcraft, J. Yang, and D. Engler. Correlation exploitation in error ranking. In *ACM SIGSOFT Software Engineering Notes*, volume 29, pages 83–93. ACM, 2004.

[19] R. Beckett, A. Gupta, R. Mahajan, and D. Walker. A general approach to network configuration verification. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 155–168, 2017.

[20] Z. Yin, X. Ma, J. Zheng, Y. Zhou, L. Bairavasundaram, and S. Pasupathy. An empirical study on configuration errors in commercial and open source systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, page 159–172, New York, NY, USA, 2011. Association for Computing Machinery.

[21] T. Xu, J. Zhang, P. Huang, J. Zheng, T. Sheng, D. Yuan, Y. Zhou, and S. Pasupathy. Do not blame users for misconfigurations. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 244–259, New York, NY, USA, 2013. Association for Computing Machinery.

[22] A. Mallidi. Encoding categorical data in machine learning. *Technical Article*, 2019.

[23] P. Bholowalia and A. Kumar. Ebk-means: A clustering technique based on elbow method and k-means in wsn. *International Journal of Computer Applications*, 105:17–24, 2014.

[24] Silhouette (clustering). https://en.wikipedia.org/wiki/Silhouette_(clustering), July 2021.

[25] Maverick: Detection of Control plane and configuration bugs using outlier analysis. https://github.com/vasu018/outlier-analyzers/, May 2021.

 ISBN: 978-1-61208-919-5