NEAt: Network Error Auto-Correct

Wenxuan Zhou Jason Croft Bingzhe Liu Matthew Caesar
University of Illinois at Urbana-Champaign
{wzhou10, croft1, bingzhe, caesar}@illinois.edu

ABSTRACT

Configuring and maintaining an enterprise network is a challenging and error-prone process. Administrators must often consider security policies from a variety of sources simultaneously, including regulatory requirements, industry standards, and to mitigate attack vectors. Erroneous implementation of a policy, however, can result in costly data breaches and intrusions. Relying on humans to discover and troubleshoot violations is slow and prone to error, considering the speed at which new attack vectors propagate and the increasing network dynamics, partly an effect of SDN. To ensure the network is always in a state consistent with the desired policies, administrators need frameworks to automatically diagnose and repair violations in real-time.

To address this problem, we present $NEAt$, a system analogous to a smartphone's autocorrect feature that enables on-the-fly repair to policy-violating updates. $NEAt$ modifies the forwarding behavior of updates to automatically repair violations of properties such as reachability, service chaining, and segmentation. $NEAt$ sits between an SDN controller and the forwarding devices, and intercepts updates proposed by SDN applications. If an update violates the policy defined by an administrator, such as reachability or segmentation, $NEAt$ transforms the update into one that complies with the policy. Unlike domain-specific languages or synthesis platforms, $NEAt$ allows enterprise networks to leverage the advanced functionality of SDN applications while simultaneously achieving strong, automated enforcement of general policies.

CCS Concepts

•**Networks** → **Network management;**

Keywords

Software-defined networking, auto-correct, real-time

1. INTRODUCTION

Modern enterprise networks must comply with highly stringent security demands, including regulatory requirements, or industry standards, such as PCI, HIPAA, and SOX. As a result, network administrators must carefully design and maintain their networks to follow those policies, by mapping out device contexts and access to sensitive resources, assessing risk, and installing access control policies that effectively mitigate that risk. However, mistakes and errors in implementing the policies can result in costly data breaches, segmentation violations, and infiltrations. Through 2020, Gartner predicts 99% of firewall breaches will be caused by misconfigurations [1, 2].

While discovering and troubleshooting these bugs is essential to maintaining network security, doing so is notoriously hard. Relying on humans to configure and maintain the network configuration is not only prone to mistakes, but slow. Given the sophistication and speed at which new attack vectors propagate, manually updating and testing new configurations leaves the network in a vulnerable state until the attack vector is fully secured. Further, maintaining a security posture in the presence of software-defined networking (SDN) is even more challenging. While SDN enables new functionality, application designers may not be aware of the policy or security requirements of the networks on which their applications will be deployed. Worse yet, SDN applications written in general-purpose languages such as Java or Python can be arbitrarily complex. Requiring applications to implement and modify their behavior to support a broad spectrum of policies needed across a broad spectrum of networks presents an almost insurmountable challenge.

To this end, we present $NEAt$, a transparent layer to automatically repair policy-violating updates in real-time. $NEAt$ secures the network with a mechanism similar to a smartphone's autocorrect feature, which enables on-the-fly repair to policy violating updates and ensures the network is always in a state consistent with policy. Unlike prior work on update synthesis, $NEAt$ maintains backward compatibility and flexibility to run general SDN application code. To do this, $NEAt$ does not synthesize network state from scratch, but rather *influences* updates from an existing SDN application toward a correct specification. In particular, $NEAt$ enforces a concrete definition of correctness by influencing and constraining dynamically arriving network instructions, in order to obey a set of specified correctness criteria. To formulate those correctness criteria, we construct a set of *policy graphs* to represent humans' correctness intent, which is based on the observation that important errors can be caught by a concise set of boundary conditions. $NEAt$ sits be-

SOSR '17, April 03-04, 2017, Santa Clara, CA, USA
 ISBN 978-1-4503-4947-5/17/04…$15.00
DOI: http://dx.doi.org/10.1145/3050220.3050238

tween an SDN controller and the forwarding devices, and intercepts the updates proposed by the running SDN applications. If the update violates an administrator's defined policy, such as reachability or segmentation, $NEAt$ transforms the update into one that complies with the policy.

A key challenge we face in this approach is discovering update repairs in real-time. In $NEAt$, we build on prior work on verification to efficiently model packet forwarding behavior as a set of Equivalence Classes (ECs) [13, 22]. Upon receiving an update from an SDN controller, $NEAt$ computes the set of affected ECs and checks for a violation in the same manner as [13]. To repair the violation, we cast the problem as an optimization problem, to find the minimum number of changes (added or deleted edges) to repair the violating EC's forwarding graph. To rapidly compute repairs on arbitrarily large networks, we propose a clustering algorithm to compress both an EC's forwarding graph and the topology, then solve the optimization problem on the compressed graphs.

A preliminary evaluation of our prototype shows promising results. On topologies with up to 125 switches and 250 hosts, $NEAt$ can discover repairs in under one second for applications with non-overlapping rules, and under two seconds for applications with more complex dependencies.

2. MOTIVATION AND DESIGN

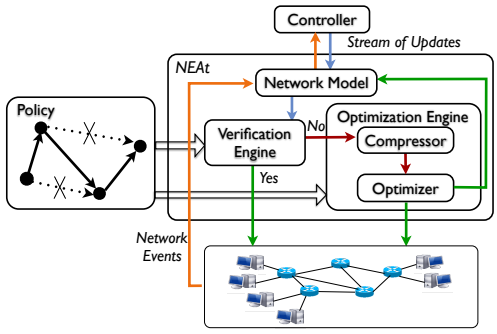


Figure 1: *System architecture of* $NEAt$.

A system to automatically repair updates, ensuring the network always remains consistent with an administrator's policy, can relieve a slow and error-prone process from the configuration process. If an update violates a given property in the network, a *repair* should fix the cause of the violation while maintaining the original purpose of the update. We argue a minimal change is best, to repair the update with the least number of added or removed edges. Furthermore, such a system should improve upon a manual effort with transparency in both architecture and performance. A system that requires hours or days to verify and repair a network is not useful if the process can be completed manually in just a few minutes. It should also not require modifying existing applications or redesigning infrastructure.

However, accomplishing this task in real-time is challenging due to the size of the network and the data plane state. To efficiently reason about the data plane, we build on previous work in verification [13, 22] that separates the forwarding behavior into Equivalence Classes (ECs) of packets. All packets within an EC are forwarded in precisely the same manner. From each EC, we can extract a *configuration graph* that defines the forwarding behavior for packets within the EC. A repair for a given EC must then explore additions or deletions of links in the configuration graph. Finding a link addition requires examining the *topology graph* defined by the edges in the physical topology. To efficiently discover repairs, we propose a clustering algorithm to compress the configuration and topology graphs, described in §3. We refer to these as the *compressed configuration graph* and *compressed topology graph*.

Figure 1 shows the overall design of $NEAt$. $NEAt$ takes as input a *policy graph*, which defines the network policies (e.g., reachability, segmentation, waypointing) in the form of a directed graph. $NEAt$ sits between the controller and forwarding devices, receiving updates from the controller, as well as updates from the network about link and switch state. With each update, $NEAt$ applies the change to a network model, from which the ECs affected by the update are computed. Using the policy graph, $NEAt$ checks each affected EC in the network model for policy violations. If the update does not introduce any violations, it is sent onto the network. However, if it does introduce a violation, the configuration graph and topology graph are compressed and passed to the optimizer. The optimizer returns a set of edges to be added or removed to the EC's configuration graph, which are then applied to the network model, converted to OpenFlow rules, and sent to the forwarding devices.

$NEAt$'s optimizer models the process of discovering repairs as an optimization problem. Our exploration of alternative approaches guided us toward this optimization problem-based solution for performance considerations. For example, consider a brute force approach that discovers repairs for a given EC by testing all possible permutations of edge additions and removals to the EC's configuration graph. A repair that requires only adding edges, from 10 possible unused topology edges, would need to explore 10! (~3.6M) permutations. If the violating property can be checked in just 1ms, each EC could take up to 10 minutes to find a repair. vfeTherefore we use the formulation described in §3 for our optimization engine.

3. ALGORITHM

We now present the core algorithm in $NEAt$ to repair violations in real-time. First, we introduce the network model and give an overview of the algorithm. Next, we describe our technique to compress the model without losing policy-relevant information. We then formulate the repair problem as an integer linear programming (ILP) problem that can be solved heuristically in real-time.

Network model As described in §2, upon intercepting an update, $NEAt$ constructs a *configuration graph* ℓ_c for each affected EC c, which captures the configured forwarding behavior for all packets in the EC c. If a violation is detected in the model, we repair it with the help of two additional graphs: a topology graph T and a policy graph $\wp$, both of which are shared across ECs. Each node in these graphs represents a physical device or a type of device, e.g., firewall, and each edge between a pair of nodes defines reacha-

bility between them. The policy graph $\wp$ is a directed graph constructed from a set of conflict-free policies that represents the expected behavior of the whole network and hence should not be violated at runtime. Figure 2 expresses a simple policy: host h should go through a firewall FW before reaching the server S. A policy's conflict freedom can be guaranteed by tools like PGA [16], which is out of the scope of this paper. A topology graph T is an undirected graph that represents the physical topology of the network.

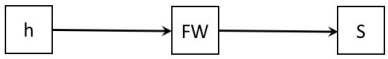


Figure 2: A simple policy graph: host h should go through a firewall FW before reaching the server S.

Algorithm overview When the verification engine discovers a violating EC, the algorithm is executed. Its goal is to repair the detected violations optimally, i.e., with the minimum number of changes to the original configuration. Upon receiving the violating EC c together with its configuration graph ℓ_c, $NEAt$ formulates the problem as an optimization problem: we aim to add or delete the minimum number of edges on ℓ_c so that the modified ℓ_c complies with $\wp_c$. $\wp_c$ is a subgraph of $\wp$ that is relevant to EC c. Note that the added or deleted edges are constrained within the topology graph T. In addition, to address the challenge of scalability, the graphs are compressed using our clustering algorithm before passing them to the optimizer. After the optimizer finds the optimal changes on the compressed graph, we map the changes back to the original graphs.

3.1 Clustering

We develop a clustering algorithm to improve scalability on arbitrarily large networks. Intuitively, the algorithm compacts configuration graphs and topology graphs into clusters around policy nodes, in a way that the resulting graphs have the same reachability properties as the original graphs with respect to nodes concerned in the policy.

For each EC c, we compress the configuration graph and the topology graph through the following three steps:

i) **Compute clusters on configuration graph around policy nodes**. For each policy node i, on configuration graph ℓ_c we perform a forward and backward breadth-first search to find node i's cluster forward tree F_i^c and backward tree B_i^c, respectively. Figure 3(b) illustrates the backward and forwarding tree for configuration graph ℓ_c. Node i's cluster C_i^c is the union of F_i^c and B_i^c. One node can belong to multiple clusters. After computing all the policy nodes, we cluster the rest of the nodes by computing F_i^c and B_i^c for each node i.

ii) **Compute clusters on topology graph.** As the configuration graph can be considered as a subgraph of the topology graph, we cluster the topology graph T based on the clusters C^c of the configuration graph. For the node that do not exist in ℓ_c, it becomes a cluster of itself, in order to give the maximum number of free nodes for the optimizer.

iii) **Compute compressed configuration and topology graph.** Let c_i denote the node i in the compressed graph for the cluster C_i^c. Any pair of nodes c_u, c_v is connected on compressed topology graph T^{cp}, if there exists an edge between any node in C_u^c and any node in C_v^c. Any pair of nodes c_u, c_v is connected with a directed edge (c_u, c_v) on compressed configuration graph ℓ^{cp}, if there exists an edge between any node in F_u^c and any node in B_v^c.

Figure 3(b) and 3(c) illustrate the clustering of a configuration graph ℓ_c and a topology graph T based on the policy graph $\wp$ in 3(a). Note that the rectangular nodes represent policy nodes. In Figure 3(b), ℓ_c is clustered around the policy nodes i, j and m. As F_i^c and B_j^c have overlapping nodes, an edge (c_i, c_j) is added to ℓ_c^{cp}. In Figure 3(c), as node q and r can not be reached by or reach any policy node, they are not compressed into any policy node cluster and hence become two separate compressed nodes in T^{cp}. Five undirected edges (c_i, c_j), (c_m, c_q), (c_q, c_j), (c_m, c_r) and (c_r, c_q) are added into T^{cp} based on physical connectivity.

3.2 Optimization

After receiving ℓ_c^{cp} and T^{cp} from the clustering phase, the optimizer determines the minimum number of edges that need to be added or deleted to make ℓ_c^{cp} consistent with $\wp_c$ using ILP.

Our integer program has a set of binary decision variables $x_{i,j,p,q}$ and $x_{i,j}$ that

$$x_{i,j,p,q}, (i,j) \in E_{T^{cp}}, (p,q) \in E_{\wp_c} \tag{1}$$

$$x_{i,j}, (i,j) \in E_{T^{cp}} \tag{2}$$

$E_{T^{cp}}$ and $E_{\wp_c}$ denote the set of edges in T^{cp} and $\wp_c$ respectively. $x_{i,j,p,q}$ defines the mappings between the edges in T^{cp} and the edges in $\wp_c$. It is 1 if a directed edge (i,j) is mapped to a policy edge (p,q) for the current EC c. $x_{i,j}$ defines the edges in the resulting configuration graph. It is 1 if a topology edge (i,j) is selected for the new configuration. These variables are required to satisfy the constraints in Equation 3-12.

For any physical link (i,j) in T^{cp} and policy link (p,q) in $\wp_c$, let $E(a)$ represent the set of all the edges of graph a, let $N(E(a))$ represent the number of all edges of graph a, and let $NB_a(i)$ represent the set of all the neighbor nodes of node i in graph a.

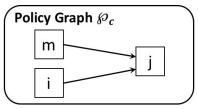
$$\forall (i,j) \qquad x_{i,j} \leq \sum_{(p,q)\in E_{\wp_c}} x_{i,j,p,q} \tag{3}$$

$$\forall (i,j) \qquad x_{i,j} \geq \sum_{(p,q)\in E_{\wp_c}} \frac{x_{i,j,p,q}}{N(E_{\wp_c})} \tag{4}$$

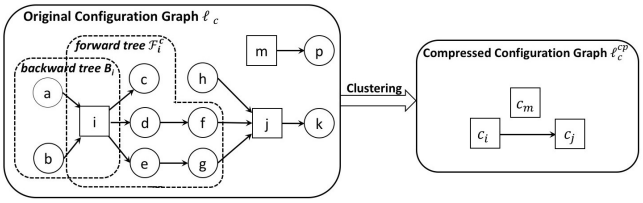
$$\forall (j,i) \qquad x_{j,i} \leq \sum_{(p,q)\in E_{\wp_c}} x_{j,i,p,q} \tag{5}$$

$$\forall (j,i) \qquad x_{j,i} \geq \sum_{(p,q)\in E_{\wp_c}} \frac{x_{j,i,p,q}}{N(E_{\wp_c})} \tag{6}$$

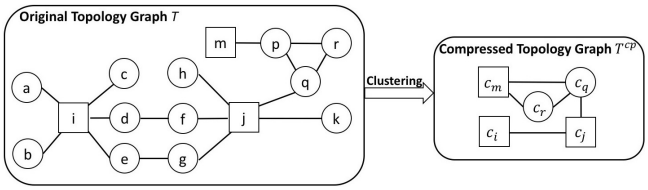
$$\forall (j,i) \qquad x_{i,j} + x_{j,i} \leq 1 \tag{7}$$



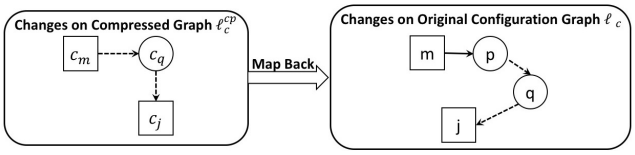
(a) Policy graph. Node i and m should be able to reach node j. Note that rectangular nodes represent policy nodes.



(b) Original and compressed configuration graphs. The reachability between m and j is violated on both graphs. The clusters are constructed around policy nodes i, j and m. $C_i^c = \{i, a, b, c, d, e, f, g\}$, $C_j^c = \{j, d, f, e, g, h, k\}$, $C_m^c = \{m, p\}$.



(c) Original and compressed topology graphs. There are two extra clusters compared with those in (b): $C_r^c = \{r\}$, $C_q^c = \{q\}$.



(d) Map back phase. Suppose the minimal change returned from optimization is the addition of edge (c_m, c_q) and (c_q, c_j), then after mapping the changes back, edge (p, q), (q, j) will be inserted to ℓ_c.

Figure 3: $NEAt$ algorithm

Equation 3 and 4 define the relationship between $x_{i,j}$ and $x_{i,j,p,q}$. They assert that any edge (i, j) in T^{cp} is selected *iff* (i, j) is mapped to at least one (p, q) in policy graph, otherwise it should not be selected. Similarly, for link (j, i), we have Equation 5 and 6. Equation 7 prevents self loops.

$$\forall i \in T^{cp} \qquad \sum_{j \in NB_{T^{cp}}(i)} x_{i,j} \leq 1 \tag{8}$$

Currently we assume unicast forwarding (Equation 8).

$\forall (p, q), \forall i \in T^{cp}$:

$$\begin{cases} \sum_{j \in NB_{T^{cp}}(i)} x_{i,j,p,q} = 1 \\ \sum_{j \in NB_{T^{cp}}(i)} x_{j,i,p,q} = 0 \end{cases} \quad \text{if } i = p \tag{9}$$

$$\begin{cases} \sum_{j \in NB_{T^{cp}}(i)} x_{i,j,p,q} = 0 \\ \sum_{j \in NB_{T^{cp}}(i)} x_{j,i,p,q} = 1 \end{cases} \quad \text{if } i = q \tag{10}$$

$$\begin{cases} \sum_{j \in NB_{T^{cp}}(i)} x_{i,j,p,q} = 0 \\ \sum_{j \in NB_{T^{cp}}(i)} x_{j,i,p,q} = 0 \end{cases} \quad \text{if } i \in \wp_c \text{ and } (\exists \rho_{i,p} \text{or } \exists \rho_{q,i}) \tag{11}$$

$$\left\{ \sum_{j \in NB_{T^{cp}}(i)} (x_{i,j,p,q} - x_{j,i,p,q}) = 0 \qquad \text{otherwise} \right. \tag{12}$$

Equations 9-12 are the flow conservation equations for policy level reachability (p, q). $\rho_{i,p}$ denotes the paths between i and p. Note that Equation 11 goes beyond individual reachability requirement in the policy, but also takes into account dependencies between policy edges. The resulting mapping is guaranteed to satisfy chaining of reachability requirements. For instance, if a policy node i is required to reach q through p, because of this equation, i cannot be mapped to the path segment (p, q). Otherwise, p might be skipped on the path from i to q.

The optimization objective is to minimize the number of changes on the original configuration graph ℓ_c.

$$\min \sum_{(i,j) \notin E_{\ell_c}} x_{i,j} \tag{13}$$

In Figure 3(a), the policy graph specifies that node m should reach j. This property is violated in the configuration graph ℓ_c in Figure 3(b). Figure 3(d) shows the optimization result on the compressed graph that two directed edges (c_m, c_q), (c_q, c_j) need to be added to repair the reachability violation between m and j. The node c_r is not chosen as it needs to add three edges (c_m, c_r), (c_r, c_q), (c_q, c_j) to repair the violation.

3.3 Map Back

The last step is to map the result to the original configuration graphs ℓ_c. The optimization result is a set of changes that consist of a set of added or deleted edges on ℓ_c^{cp}. While mapping back, an edge (c_i, c_j) becomes a path between the cluster C_i^c and C_j^c. If an edge (c_i, c_j) is added to ℓ_c^{cp}, all edges along the shortest path $P \in \rho_{C_i^c, C_j^c}$ in T should be added into ℓ_c. If an edge (c_i, c_j) should be removed from ℓ_c^{cp}, any path $P \in \rho_{C_i^c, C_j^c}$ in ℓ_c needs to be removed.

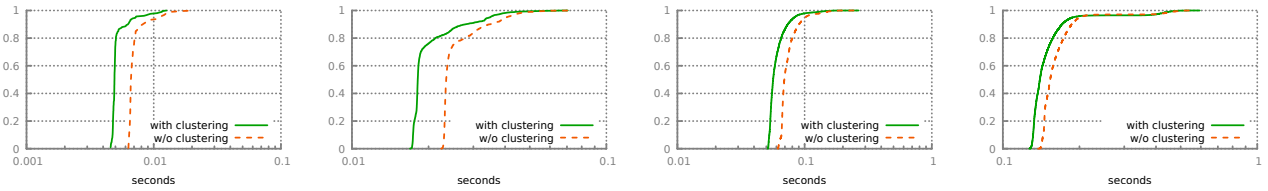
Figure 3(d) illustrates the map back phase. As described in §3.2, the optimization result consists of two added edge: (c_m, c_q) and (c_q, c_j). In Figure 3(c), we can see that in T the shortest path between c_m and c_q is (p, q), and between c_q and c_j is (q, j). The path $\{(p, r), (r, q)\}$ should not be chosen for (c_m, c_q) as it is not the shortest path between the two clusters. The two dotted arrows (p, q) and (q, j) in 3(d) illustrate the edges that need to be added into ℓ_c.

Afterward, these computed changes will be translated into SDN instructions, and sent to the network devices.

4. EVALUATION

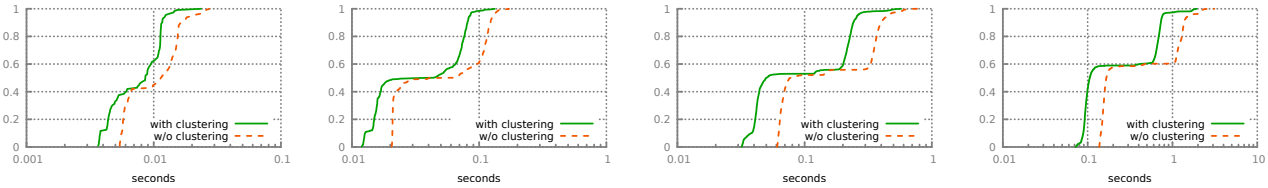
We implemented a prototype of $NEAt$ in Python. The verification engine is based on our prior work [13] and we use the Gurobi Optimizer [3] within our optimization engine to solve the ILP.

To evaluate the feasibility and scalability of $NEAt$, we synthesized a set of fat-tree topologies with various sizes, and used $NEAt$ to maintain reachability and segmentation policies. More specifically, on each topology, under random



(a) 16 hosts, 20 switches (b) 54 hosts, 45 switches (c) 128 hosts, 80 switches (d) 240 hosts, 125 switches

Figure 4: ***Repair time comparison under random removals of exactly matching rules***



(a) 16 hosts, 20 switches (b) 54 hosts, 45 switches (c) 128 hosts, 80 switches (d) 240 hosts, 125 switches

Figure 5: ***Repair time comparison under random removals of overlapping rules***

removals of rules, we measured the repair time for each removal that caused a violation.

4.1 Exact matching rules

We first focus on flow-based traffic management applications widely used in SDN [5, 8, 10, 11, 12]. Any forwarding rule produced by such applications at a switch matches at most one flow. In our terms, each rule only affects at most one EC. Suppose the desired policy is that every pair of hosts should be able to reach each other, which we will refer to as an *all-pair reachability* policy. With random removals of rules, for those removals resulting in violations to *all-pair reachability*, the optimization engine is triggered to perform the repair. For testing purposes, we re-check the policy after each repair, and the check passed for all cases. Figure 4 compares the CDFs of the repair time (including clustering and map-back time, when using clustering) with and without clustering over rule removals under various topology sizes. We can see the repair time is bounded under one second, and clustering brings constant speed-up to optimization.

4.2 Overlapping rules

For networks that use wild-carded rules or longest prefix matching, the assumption in the previous subsection does not hold. One rule may affect multiple ECs, and thus potentially trigger repairs on multiple graphs. Fortunately, there is a trend to move such overlapping rules to network edge or even hosts [4, 6, 14], leaving the core with exactly matching rules. In order to study how $NEAt$ performs under this less preferable but less common scenario, we assign IP addresses within the same prefix subnet to hosts within the same pod on the fat-tree topologies. We then compress rules on the switches as much as possible. For example, each core switch has only k forwarding rules, where k is the number of pods, and each rule matches on one pod's prefix. Similar to the previous experiments, we used $NEAt$ to guarantee an *all-pair reachability* policy, and our engine discovered repairs for all violations. Figure 5 again compares the CDFs of the repair time with and without clustering triggered. The repair took longer compared to applications with exact match rules because of the increased number of affected ECs. With our clustering technique, optimization is able to finish under two seconds in the worst case.

4.3 Segmentation policy

Lastly, we tested how $NEAt$ handles a mix of reachability and segmentation policies. For each fat-tree topology, we randomly selected a pair of pods whose hosts should be isolated from each other. In addition, any other pair of pods should have bi-directional reachability. Every repaired configuration was checked against the policy and passed the check, with speed comparable to previous subsections. Note that unlike the previous pure reachability policy, where repairs are all edge additions, in this case, a repair is sometimes a mix of edge additions and deletions. As verified by the re-checks, changes for fixing different types of policies keep other policy intact.

5. RELATED WORK

SDN programming languages: Many programming languages have been proposed to provide abstractions to program SDNs, e.g., Frenetic [7], Pyretic [17] and Maple [21]. These allow programmers to compose complex rules without the need to manually resolve conflicts between rules. However, these languages face limitations implementing general policies that deliver higher-level intent, such as expressing middleware functionality or QoS constraints.

SDN synthesis platforms: Network state can also be synthesized from a set of pre-specified correctness conditions. NetGen [19], for example, takes as input a specification using regular expressions to define paths changes and a set of ECs to modify. It uses an SMT solver to find the minimal number of changes. However, similar to Merlin [20] and FatTire [18], this tool is designed to be used as compiler, with performance that is too slow for real-time applications (i.e., minute-scale synthesis). While using NetGen in place of our ILP is possible, doing so would additionally require translating each update into an equivalent Net-

Gen specification. Similarly, Marham [9] proposes a framework for automated repair, but with slower performance — on the order of several seconds for topologies 10s of nodes and links. Margrave [15] analyzes changes to access control policy changes, highlighting to an operator the effect it has on the policy, without suggesting repairs to violtions.

6. DISCUSSION AND FUTURE WORK

With this proposed technique, we are facing the following five key challenges that lead to corresponding future work.

Theoretic Proof The design of $NEAt$ is centered on the key word "correctness". A natural question is how $NEAt$ ensures correctness. Intuitively, the network changes output by $NEAt$ should always be correct with regard to the desired policies in the network, because the repair procedure is actually modifying network updates to make the new forwarding graph comply with the policy graph, i.e., the policy graph can be mapped to the forwarding graph. However, as the repair procedure involves solving an optimization problem, which is not guaranteed to be feasible in every case, without a safe fall-back plan we cannot ensure $NEAt$ is always correct. Moreover, even with a feasible solution, the solution may not be the best, in our terms with the minimum number of changes, due to the relaxation on the ILP problem made by our chosen solver. In addition, the repair effort uses a minimal number of edits as the optimization goal. In practice, there may be other goals, for example, minimizing the amount of traffic shifts, etc. We plan to carefully consider all these factors, and prove, both theoretically and empirically, how accurate $NEAt$'s solution is under different scenarios, and under what types of scenarios $NEAt$ is applicable.

Interaction with Applications Upon detecting a violation, $NEAt$ has two options: allowing the network operator to review one (or more possible) repairs before it is applied to the network, or automatically fixing it. $NEAt$ proposes removing the human element as much as possible. The first case is against our decision and reintroduces the possibility of human error. Furthermore, as networks become more dynamic (e.g., an SDN traffic engineering application that issues updates with each new flow), repairs requiring manual review may arrive faster than an operator can process them. In the second case, however, fixes are applied silently, without informing upper-layer applications and operators. Consider a load balancer application, which balances traffic across the network by tracking current load and routes in the network. If $NEAt$ directly fixes a violation, the state in the application will become inconsistent from the network state.

As a result, we plan to deploy $NEAt$ with two modes. In the first, $NEAt$ acts as a transparent layer between an SDN controller and network devices, automatically repairing violations without operator intervention. Essentially, $NEAt$ is equivalent to an application that overrides all other applications. We argue this is best suited for stateless applications. Alternatively, $NEAt$ can be integrated within SDN applications as a library. Here, $NEAt$ can be used by applications to process queries that include proposed updates by the application, and application-level intentions. The results of the query include whether or not the updates violate any network policies, and if so, what is the modified set of updates. Note that the modified updates should be computed in a way that take into account the application intention.

There are clear trade-offs between the two modes. The first directly takes advantage of our prior work on data plane verification, but will cause network states to diverge from application views. The second one retains consistency between network state and application views, but loses the ability to analyze the network in an unified way. We plan to test these two approaches with real network traces and applications.

Generality and Expressiveness of Correctness Policies Currently, we focus on network policies that fall into *trace properties* which we studied in [22]. Such properties characterize the paths that packets traverse through the network. Fortunately, this covers many common network properties, including reachability, access control, loop freedom, and waypointing. In addition, as such properties are path-based, composing them into a policy graph is relatively straightforward. However, there are other network operational goals, for example, ensuring shortest paths or at lease k link disjoint paths between given devices. To address that, we will first extend $NEAt$ with extensive APIs to code general reachability policies with. Next, we will investigate techniques to compose such a broader range of policies, and then incorporate those policies into our optimization procedure.

Performance Optimization In our optimization formulation, the number of variables for one EC is approximately the product of the number of edges in the physical topology and the number of edges of the policy graph. This can easily exceed 100k. Therefore, we need a more scalable formulation. To this end, we are currently studying policy-preserving graph compression algorithms, to reduce the size of the optimization problem. We are also exploring optimization techniques to speed up $NEAt$.

Evaluation using Realistic Network Data Plane Traces and Applications Future work will build $NEAt$ into a fully-featured SDN data plane autocorrect platform. We will test $NEAt$ with real-world data traces and SDN applications to validate this approach. We expect the investigation will bring us insights on how to improve the design.

7. CONCLUSION

In this paper we presented $NEAt$, a system that provides network administrators with a network analogue of a smartphone's autocorrect. As a transparent layer, $NEAt$ repairs, in real-time, updates from an SDN controller that violate generic policies such as reachability, service-chaining, and segmentation. $NEAt$ repairs the updates by adding or removing a minimal number of rules in order to comply with the policy by casting the problem as an optimization problem. Preliminary experiments on large fat-tree topologies show our optimization problem-based formulation can discover repairs in under one second for applications with non-overlapping rules, and two seconds for applications issuing rules with more complex dependencies.

References

[1] http://www.infosecurity-magazine.com/opinions/to-err-is-human-to-automate-divine/.

[2] http://www.verizonenterprise.com/verizon-insights-lab/dbir/2016/.

[3] Gurobi optimization. http://www.gurobi.com/.

[4] Network virtualization for cloud data centers. http://tinyurl.com/c9jbkuu.

[5] T. Benson, A. Anand, A. Akella, and M. Zhang. Microte: Fine grained traffic engineering for data centers. In *Proceedings of the Seventh Conference on Emerging Networking Experiments and Technologies-periments and Technologiesnference on emerging Networking EXperiments and Technologies (CoNEXT)*, page 8. ACM, 2011.

[6] B.Raghavan, M.Casado, T.Koponen, S.Ratnasamy, and a. S. S. A.Ghodsi. Software-defined Internet architecture: Decoupling architecture from infrastructure. In *HotNets*, 2012.

[7] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker. Frenetic: A network programming language. In *ICFP*, 2011.

[8] B. Heller, S. Seetharaman, P. Mahadevan, Y. Yiakoumis, P. Sharma, S. Banerjee, and N. McKeown. ElasticTree: Saving energy in data center networks. In *NSDI*, 2010.

[9] H. Hojjat, P. Reummer, J. McClurgh, P. Cerny, and N. Foster. Optimizing horn solvers for network repair. In *FMCAD*, 2016.

[10] C.-Y. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, and R. Wattenhofer. Achieving high utilization with software-driven wan. In *SIGCOMM*, 2013.

[11] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. HoÌĹlzle, S. Stuart, and A. Vahdat. B4: Experience with a globally-deployed software defined wan. In *SIGCOMM*, 2013.

[12] X. Jin, R. Mahajan, H. H. Liu, R. Gandhi, S. Kandula, M. Zhang, J. Rexford, and R. Wattenhofer. Dynamic scheduling of network updates. In *SIGCOMM*, 2014.

[13] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey. VeriFlow: Verifying network-wide invariants in real time. In *NSDI*, 2013.

[14] M.Casado, T.Koponen, S.Shenker, and A.Tootoonchian. Fabric: A retrospective on evolving sdn. In *HotSDN*, 2012.

[15] T. Nelson, C. Barratt, D. J. Dougherty, K. Fisler, and S. Krishnamurthi. The margrave tool for firewall analysis. In *LISA*, 2010.

[16] C. Prakash, J. Lee, Y. Turner, J.-M. Kang, A. Akella, S. Banerjee, C. Clark, Y. Ma, P. Sharma, and Y. Zhang. PGA: Using graphs to express and automatically reconcile network policies. In *SIGCOMM*, 2015.

[17] J. Reich, C. Monsanto, N. Foster, J. Rexford, and D. Walker. Modular sdn programming with pyretic. In *USENIX ;login, 38(5)*, pages 40–47, October 2013.

[18] M. Reitblatt, M. Canini, A. Guha, and N. Foster. Fattire: Declarative fault tolerance for software-defined networks. In *HotSDN*, 2013.

[19] S. Saha, S. Prabhu, and P. Madhusudan. NetGen: Synthesizing data-plane configurations for network policies. In *SOSR*, 2015.

[20] R. Soule, S. Basu, P. J. Marandi, F. Pedone, R. Kleinberg, E. G. Sirer, and N. Foster. Merlin: A language for provisioning network resources. In *CoNEXT*, 2014.

[21] A. Voellmy, J. Wang, Y. R. Yang, B. Ford, and P. Hudak. Maple: Simplifying sdn programming using algorithmic policies. In *SIGCOMM*, 2013.

[22] W. Zhou, D. Jin, J. Croft, M. Caesar, and P. B. Godfrey. Enforcing customizable consistency properties in software-defined networks. In *NSDI*, 2015.