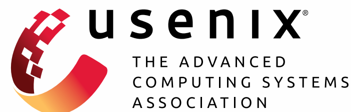


Automatically Correcting Networks with *NEAt*

Wenxuan Zhou, Jason Croft, Bingzhe Liu, Elaine Ang,
and Matthew Caesar, *University of Illinois at Urbana-Champaign*

https://www.usenix.org/conference/nsdi18/presentation/zhou

This paper is included in the Proceedings of the
15th USENIX Symposium on Networked
Systems Design and Implementation (NSDI '18).
April 9–11, 2018 • Renton, WA, USA
ISBN 978-1-939133-01-4

Open access to the Proceedings of
the 15th USENIX Symposium on Networked
Systems Design and Implementation
is sponsored by USENIX.

Automatically Correcting Networks with *NEAt*

Wenxuan Zhou, Jason Croft, Bingzhe Liu, Elaine Ang, Matthew Caesar
University of Illinois at Urbana-Champaign
{*wzhou10, croft1, bingzhe, ranang2, caesar*}*@illinois.edu*

Abstract

Configuring and maintaining an enterprise network is a challenging and error-prone process. Administrators often need to consider security policies from a variety of sources such as regulatory requirements, industry standards, and mitigating attack vectors. Erroneous configuration or network application could violet crucial policies, and 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 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. It does so by modifying the forwarding behavior of updates to automatically repair violations of policies such as reachability, service chaining, and segmentation. *NEAt* takes as input a set of administrator-defined high-level policies, and formulates these policies as directed graphs. Sitting between an SDN controller and the forwarding devices, *NEAt* intercepts updates proposed by SDN applications. If an update violates a policy, *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. Based on a prototype implementation and experimentation using Mininet and operation trace of a large enterprise network we demonstrate that *NEAt* achieves promising performance in real-time bug-fixing.

1 Introduction

Modern enterprise networks must comply with highly stringent security demands that merge together demands from a variety of sources and standards. As a result, network administrators must carefully design and maintain their networks to follow these 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. 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 error conditions can be caught by a concise set of boundary conditions. *NEAt* sits between 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 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) [19, 30]. 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 [19]. 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 exploit two optimization techniques, *topology limitation* which "slices" away irrelevant part of the network, and *graph compression*, to compress both an EC's forwarding graph and the topology. Then we solve the optimization problem on the sliced and compressed graphs.

Furthermore, as *NEAt* repairs policy-violating updates, stateful applications — without knowledge of the violating or repaired updates — will diverge from the underlying network state. To address this problem, applications can interactively propose updates to *NEAt* and receive notifications of repairs with minor modifications to application code. Thus, applications can remain unmodified and leverage *NEAt* transparently in a *pass-through* mode, with a risk of state divergence, or propose updates in an *interactive* mode.

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. Furthermore, we find *NEAt* can verify and repair updates on realistic data planes. On a large enterprise network with 1M forwarding rules, *NEAt* discovered and repaired 28 loop violations. Simulations on this data set show *NEAt* can verify and repair reachability and loop freedom policies in under a second.

2 Motivation

Enterprise network policies must compose together requirements from a variety of demands to mitigate risk for attack vectors and limit access to sensitive resources. As a result, network administrators must take into account complex, composed policies configuring or updating a network. This is a slow and often error-prone process for a human operator. The operator may introduce errors translating the demands into high-level policies, or translating the policies into low-level routing configurations. While tools [17, 19] exist to automatically discover misconfigurations in real-time, they offer the operator no guidance on how to repair the misconfiguration beyond the type of correctness property that is violated. Rather, these tools block updates from introducing violations into the data plane state, at the cost of functionality.

Instead, a system to automatically repair updates, ensuring the network always remains consistent with the 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.

Efficiently discovering repairs is not a trivial addition on top of data plane verification tools, such as [17, 19]. Due to the size of the network and data plane state, performance is a key challenge in repairing policy violations in real-time. Consider a naive approach built on top of VeriFlow that separates the forwarding behavior into Equivalence Classes (ECs) of packets. All packets within an EC are forwarded in precisely the same manner. Each EC defines a *configuration graph* that captures the the forwarding packets for packets within the EC. The number of ECs is dependent on the number of devices and forwarding rules in the network, and the time to discover a repair is dependent on the number of ECs and the number of edges in the network. A brute force approach might discover repairs by testing all permutations of edge additions and removals to an 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.

3 Design

NEAt operates between the controller and switches, intercepting and verifying updates against a set of correctness properties specified by a network operator. *NEAt* takes these properties as input in the form of a directed graph called a *policy graph* (① in Figure 1). Policy graphs can express properties including reachability, segmentation, and waypointing, as described in §4.

To verify updates conform to the operator's intended policies, each update is applied to a model of the data plane state and checked using *NEAt*'s verification engine. a policy violation, the correction engine transforms the update or existing data plane state to satisfy the violated policy. This ensures only updates conforming to

the network policies are sent onto the network.

NEAt can integrate with the existing SDN control infrastructure in two ways. It can serve as a transparent layer in a *pass-through mode*, or can interact with controller applications in an *interactive mode* with minor changes to the applications.

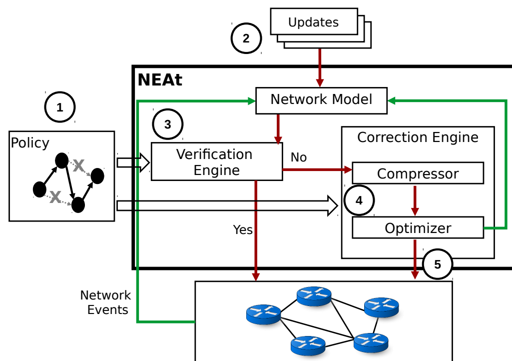


Figure 1: *System architecture of NEAt.*

3.1 Verification and Repair

To verify updates and efficiently reason about the current data plane states, *NEAt* builds on our previous work in data plane verification [19,30] that separates the forwarding behavior into Equivalence Classes (ECs) of packets. 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 two optimization techniques to compress the configuration and topology graphs, described in §6. We refer to the outcome of these techniques as the *compressed configuration graph* and *compressed topology graph*.

With each update (②), *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 using the verification engine (③). 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 correction engine (④). 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 (⑤).

3.2 Interaction Modes

To prevent applications from diverging from the underlying network state, *NEAt* exposes two integration modes: *pass-through* and *interactive*.

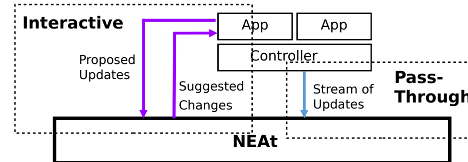


Figure 2: *Interaction modes of NEAt.*

In pass-through mode, *NEAt* acts as a transparent layer that sits between the controller and forwarding devices. This mode enforces network policies without modification to controller or SDN applications. Both these applications and the controller are unaware of *NEAt*. *NEAt* intercepts updates from the controller, as well as updates from the network about link and switch state, and passes them to the verification and correction engines. The corrected updates abide by correct network policies, and are directly applied to the network.

It's possible for applications to have a different view with *NEAt* about the current network state in pass-through mode. But this does not violate consistency, since *NEAt* acts as an arbitrator during rule insertion, and will diligently verify and correct updates regardless of the application's intention. The original intention of an application is well preserved if the application is written with full knowledge of, and in accordance with, the network-wide correctness criteria. Otherwise, *NEAt* may sacrifice application correctness for the benefit of enforcing correct network policies.

Interactive mode enables applications to leverage *NEAt*'s verification and repair process by checking proposed updates. An application passes to *NEAt* a set of updates, which are checked against the current network model. If the updates introduce a violation, *NEAt* returns a set of repaired updates, which the application can accept or reject. If the application accepts the changes, it can send them onto the network and update its state, ensuring the application and network state are consistent. If the application rejects the changes, it can propose another set of updates to *NEAt*. Interactive mode requires modifications to applications to update its state with the accepted changes.

NEAt maintains consistency between the interaction modes, allowing applications and the controller to both simultaneously benefit from *NEAt*'s automated repair. For example, one application can use *NEAt*'s API while another remains unmodified, allowing its updates to be checked by *NEAt* in pass-through mode.

4 Policy as Graphs

Many existing tools reason about individual network paths [18, 19]. While this approach has proven effective for network verification, synthesizing network state changes requires viewing the entire network as a whole

(i.e., a graph), as changes that repair one path may influence the correctness of other paths. In addition, expressing network correctness conditions as a graph instead of a collection of paths enables dealing with a richer set of policies, for instance, path consistency and load balancing. Based on this intuition, *NEAt* takes as input a set of intended policies, and formulates these policies as directed graphs called *policy graphs*.

A *policy graph* is defined on a packet header pattern, for example, ip dst 10.0.1.0/24, port 443. Each node on a policy graph is a traffic footprint matching a particular packet header pattern at a certain network location, e.g., a switch, a routing table, an ACL table. Edges are marked with labels denoting different types of reachability constraints. For example, the graph in Figure 3 requires that at least m paths exist from node A to B when $m > 0$, each bounded by n hops, or no path exists from A to B when $m = 0$. For simplicity, packet header patterns are not depicted. Next, we show how to represent several commonly-used network policies as policy graphs.

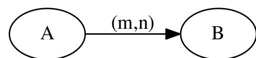


Figure 3: *Policy edge*

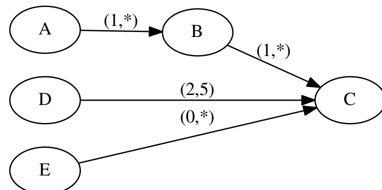


Figure 4: *Policy graph*

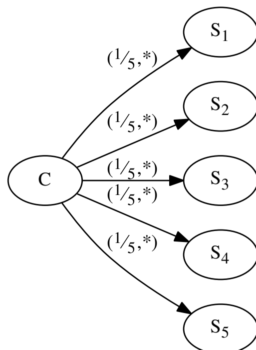


Figure 5: *Load balancing policy*

Reachability This policy requires that at least one path exists from one node to another. It is expressed with $m = 1$ and n unspecified (shown as "*"), for example, the edges $A \rightarrow B$ and $B \rightarrow C$ in Figure 4.

Bounded path length This policy defines the maximum number of hops between two reachable nodes. The path length is specified using n. For instance, the path from D to C in Figure 4 is bounded by 5 hops.

Shortest path This policy can be viewed as a special case of the bounded path length policy, where each path is bounded by the length of the corresponding shortest path in the topology. Therefore, it can be encoded in a similar way as bounded path length policy.

Multipath This policy requires multiple paths exist from one node to another. It is expressed by assigning m an integer larger than one. As shown in Figure 4, there should be at least two paths from D to C ($m = 2$).

Isolation This policy prevents one node from reaching another, which is expressed by specifying $m = 0$ on the edge connecting the two isolated nodes.

Service chaining The policy defines a set of *waypoints* that one flow must traverse in sequence. It is represented by concatenating edges on a policy graph. For example, in Figure 4, traffic from node A should traverse a waypoint B before reaching C.

Load balancing This policy requires distributing traffic from a source to a pool of servers according to a specified distribution. In our policy model, it is expressed by assigning m a fractional value . As an example, Figure 5 denotes a policy that requires traffic from client C to be distributed evenly among five servers.

In summary, a policy graph is able to express both qualitative and quantitative reachability constraints.

5 Repair Algorithm

In this section, we present *NEAt*'s core algorithm for repairing violations at runtime constrained by a given policy graph. First, we introduce the network model and give an overview of the algorithm. Next, we describe our formulation of the repair problem for basic reachability policies as an integer linear programming (ILP) problem. We then generalize this approach to repair the wider range of policies discussed in 4.

Network Model As described in §3, upon intercepting an update, *NEAt* constructs a network graph model for each affected EC that captures the configure forwarding behavior for all packets within the EC. This directed *configuration graph* ℓ_c, along with a topology graph T and policy graph $\wp$ serve as inputs to the repair algorithm.

Each node in these graphs represents a host or a networking device, and each edge between a pair of nodes defines reachability between them. The policy graph $\wp$, as discussed in 4, 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. Policy conflict freedom can be guaranteed by tools like PGA [23], 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.

Algorithm Overview When the verification engine discovers a violated 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. *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 edges are constrained within the topology graph T. We solve the optimization problem using ILP.

Subsection §5.1 describes the repair algorithm for basic reachability policies, and subsection §5.2 enhances the basic algorithm to cope with the entire set of policies in §4. We complete the section with our repair algorithm for forwarding loops (§5.3). Table 1 summarizes the key notations used in this section and the next section §6.

Symbol	*Description*
ℓ_c	The configuration graph for EC c.
$\wp$	The policy graph.
T	The topology graph.
(i,j)	The edge from node i to node j.
ρ_{ij}	The paths between node i and node j.
C_i^c	The cluster of node i for equivalence class c.
c_i	The compressed node i for C_i^c.
E_a	The set of all edges in graph a.
$N(E_a)$	Number of all edges in graph a.
$NB_a(i)$	The set of neighbors of node i in graph a.

Table 1: Key notations in problem formulation.

5.1 Repair Basic Reachability

We start with the basic case where $\wp_c$ contains only reachability constraints. Our integer program has a set of binary decision variables $x_{i,j,p,q}$ and $x_{i,j}$ where

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

$$x_{i,j}, (i,j) \in E_T \tag{2}$$

E_T and $E_{\wp_c}$ denote the set of all edges in T and $\wp_c$ respectively. Variable $x_{i,j,p,q}$ defines the mapping between a physical edge and a policy graph edge. It is one if a directed edge (i,j) is mapped to policy edge (p,q) for the current EC c, i.e., the flow from p to q will be forwarded through edge (i,j) from i to j. Variable $x_{i,j}$ defines whether edge (i,j) is used for forwarding this EC's traffic regardless of which flow uses it. Edge (i,j) in T is selected if any flow (p,q) is forwarded through (i,j) (Equation 3). Similarly, for the other direction (j,i), we have Equation 4. No physical link can be selected to forward traffic for the same EC on both directions (Equation 5) to avoid loops.

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

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

$$\forall(j,i) \qquad x_{i,j} + x_{j,i} \leq 1 \tag{5}$$

Equations 6-8 are the flow conservation equations for policy level reachability (p,q). $\forall(p,q), \forall i \in T$:

$$\begin{cases} \sum_{j\in NB_T(i)} x_{i,j,p,q} = 1 \\ \sum_{j\in NB_T(i)} x_{j,i,p,q} = 0 \end{cases} \quad \text{if } i = p \tag{6}$$

$$\begin{cases} \sum_{j\in NB_T(i)} x_{i,j,p,q} = 0 \\ \sum_{j\in NB_T(i)} x_{j,i,p,q} = 1 \end{cases} \quad \text{if } i = q \tag{7}$$

$$\left\{ \sum_{j\in NB_T(i)} (x_{i,j,p,q} - x_{j,i,p,q}) = 0 \right. \quad \text{otherwise} \tag{8}$$

The optimization objective is to minimize the number of changes (additions and deletions) on the original configuration graph ℓ_c.

$$\min \left(\sum_{(i,j)\notin E_{\ell_c}} x_{i,j} - \sum_{(i,j)\in E_{\ell_c}} x_{i,j} \right) \tag{9}$$

5.2 Generalizing the Algorithm

To support generalized reachability policies in §4, we encode several additional constraints into the ILP.

Isolation We introduce a special *DROP* node. If two nodes are required to be isolated, i.e., the nodes are connected with a $(0,*)$ edge in the policy graph, we change the way flow conservation equations are defined. In particular, we replace Equation 7 with Equations 10 and 11 below in the flow conservation equations. That is, a flow from p to q should sink at *DROP* before reaching q.

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

$$\left\{ \sum_{j\in NB_T(i)} x_{i,j,p,q} = 0 \right. \quad \text{if } i = q \tag{11}$$

Service Chaining With service chaining, or waypointing, we enhance our flow conservation equations with Equation 12. It extends the definition beyond individual reachability segments (policy graph edges), by taking 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, node i in the configuration graph is not allowed to carry flow from p to q. Without this equation, p might be skipped on the path from i to q.

$$\begin{cases} \sum_{j\in NB_T(i)} x_{i,j,p,q} = 0 \\ \sum_{j\in NB_T(i)} x_{j,i,p,q} = 0 \end{cases} \quad \text{if } i \in \wp_c \text{ and } (\exists\rho_{i,p} \text{or } \exists\rho_{q,i}) \tag{12}$$

Bounded or Equal Path Length/Shortest Path If a path length bound n is specified for a policy edge (p,q), then a new constraint is added (Equation 13):

$$\sum_{(i,j)\in E_T} (x_{i,j,p,q} + x_{j,i,p,q}) \leq n \tag{13}$$

Multipath If at least m link-disjoint paths are required for flow (p,q), then the flow conservation equations 6 and 7 are updated as Equation 14 and 15 respectively. Multipath requirements are enforced throughout the distance between two end nodes by Equation 8.

$$\begin{cases} \sum_{j \in NB_T(i)} x_{i,j,p,q} \geq m \\ \sum_{j \in NB_T(i)} x_{j,i,p,q} = 0 \end{cases} \quad \text{if } i = p \qquad (14)$$

$$\begin{cases} \sum_{j \in NB_T(i)} x_{i,j,p,q} = 0 \\ \sum_{j \in NB_T(i)} x_{j,i,p,q} \geq m \end{cases} \quad \text{if } i = q \qquad (15)$$

Load Balancing As discussed in §4, policy edges within a load balancing policy are denoted with a decimal path count. Correspondingly, in our optimization problem, variables that map physical edges to policy edges are also decimal values between zero and one, instead of binary values. In addition, we introduce a new equation (Equation 16) to capture how flow distribution propagates.

$$\prod_{x_{i,j,p,q} \neq 0} x_{i,j,p,q} = m \qquad (16)$$

For example, consider the network in Figure 6, where there are two layers of load balancing between client C and servers $S1$ $S5$. If the policy in Figure 5 is required, the solutions for variables $(x_{i,j})$ are shown in Figure 6.

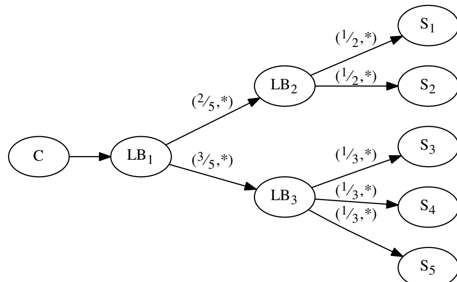


Figure 6: *Load Balancing Configuration.*

5.3 Repairing Loops

The preceding repair algorithm operates on a *loop-free* configuration graph. As such, we first check for and remove loops from each configuration graph before compressing and repairing violations of any other property type. Our objective for repairing loops is to minimize change to the network, with a preference to affect few equivalence classes as possible, as well as removed the minimal number of rules. Thus, our algorithm will remove a forwarding rule matching packets destined to 10.0.0.1/32 over one for 10.0.0.0/8. Since loops are repaired first, and *NEAt* will later check reachability properties on each equivalence class, our loop repair algorithm does not need to consider introducing permanent reachability violations by removing rules.

Algorithm 1 presents our loop repair algorithm. $\Theta(c)$ denotes the set of all loops appearing in a configuration graph ℓ_c and $N(\Theta(c))$ the number of loops in ℓ_c. θ_i is a subgraph of ℓ_c, and $N(\theta_i) = 0$ when the subgraph contains no loops. The algorithm begins by finding and removing all intersecting edges across ℓ_c's loops. For each loop in ℓ_c that is not repaired by removing these edges, next remove an edge (i, j) where i's IP address is the destination, if such an edge exists. While θ_i still has loops, remove an edge in the loop which has the most specific match rule (e.g., longest prefix). Each edge is mapped to a specific forwarding rule at a particular switch when we compute the equivalence classes.

Algorithm 1 Loop repair

procedure REMOVELOOP(ℓ_c, $\Theta(c)$)
    # remove edges appearing in multiple loops
    remove $\{(i, j) \mid (i, j) \in \theta_k \wedge (i, j) \in \theta_m \forall k,m \in \Theta(c)\}$
    if $N(\Theta(c)) = 0$ **then**
        return ℓ_c
    for all $\theta_i \in \Theta(c)$ **do**
        while $N(\theta_i) > 0$ **do**
            # remove edges forwarded out the destination
            remove (i, j) if i is destination
        while $N(\theta_i) > 0$ **do**
            # remove most specific forwarding rule
            remove $(i, j) \in \theta_i$ with longest prefix
    return ℓ_c

Removal of a forwarding rule is accomplished by replace it with a drop rule, to prevent a coarser match from introducing another loop. For example, if a rule matching destination IP 10.0.0.1/32 is simply deleted from a switch's forwarding table, another rule matching 10.0.0.1/31 on the same switch and forwarding to the same next hop could prevent the loop from being repaired. To conserve switch memory during in response repairs, *NEAt* checks all coarser drop rules to determine if multiple rules can be aggregated together.

6 Optimizations

While conceptually straightforward, the repair algorithm in section 5 does not scale to well. In the optimization problem formulation, the number of variables for one EC is approximately the product of the number of topology links and the number of policy graph edges, which can easily exceed 100k. In this section, we present two techniques that dramatically optimize the repair speed.

6.1 Topology Limitation

This technique aims to "slice" away irrelevant or redundant part of the network, and thus shrink the size of the optimization problem. After getting a configuration graph that violates some policies, before passing it to the optimizer, we first remove disconnected components on

the physical topology. Next, we localize the potential affected area on the topology. Fortunately, most modern networks are designed in a hierarchical structure. Examples include data centers arranged in a fattree topology, and enterprise networks divided into multiple sites joint by a backbone network. Such a structure implies certain communication pattern: communication within a subtree should stay local, for example, and communication between subtrees normally doesn't traverse other subtrees, i.e., go through a valley. In our linear programming problem, typically only a subset of the topology edges is considered mappable to a policy edge. Results in section 8 shows the effectiveness of this technique.

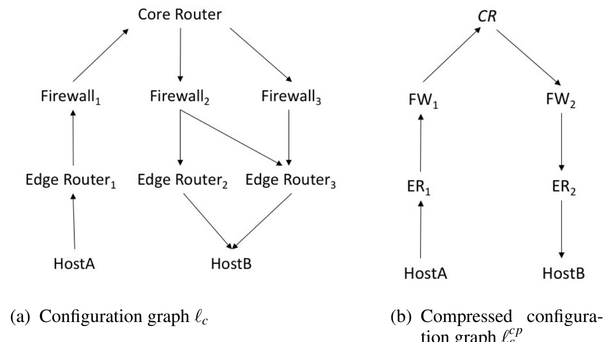
6.2 Graph Compression

Besides hierarchical structures, most large networks are designed in patterns that enforce symmetry to some extent [22] for load balancing or resilience reasons. For example, in a data center fattree topology, devices on the same layer (access, aggregate, core) are symmetrically connected to multiple devices on the neighboring layers. We exploit such regularities to compress the graphs. The key to the compression is that the compressed graphs must be equivalent to the original graphs with respect to the policies of interest. To this end, we leverage a graph pattern preserving compression [10] as the major building block of *NEAt*'s compressor (Figure 1). The algorithm compresses a labeled directed graph according to the following bisimulation relation:

Bisimulation Relation [9] We denote $G = (V,E,L)$ as a labeled directed graph. V represents a set of node and $(u,v) \in E$ represents a directed edge from node u to node v. $L(u) \in \Gamma$ represents the label of node u, where Γ is the set of labels that applied to V. In the networked system context, the labels may represent a set of similar functional networking nodes, e.g. hosts, firewalls, load balancers. For example, in Figure 7(a), we label the network nodes as *Firewall*, *Edge Router* and *Core Router* and we label the two hosts as $HostA$ and $HostB$.

A *bisimulation relation* on a graph $G = (V,E,L)$ is a binary relation $BR \subseteq V \times V$ such that for all $(u,v) \in BR$:
(a) $L(u) = L(v)$;
(b) $\forall(u,u') \in E, \exists(v,v') \in E$ such that $(u',v') \in BR$;
(c) $\forall(v,v') \in E, \exists(u,u') \in E$ such that $(u',v') \in BR$.
In Figure 7(a), $Firewall_2$ and $Firewall_3$ are *bisimular* to each other, while $Firewall_1$ is not *bisimular* to any other firewall. Because $HostB$ is solely in a *bisimular* cluster, and hence $EdgeRouter_1$ and $EdgeRouter_2$ are *bisimular* as they only has one child $HostB$. As $Firewall_2$ and $Firewall_3$ have the children that are *bisimular*, they are also *bisimular* to each other. While $Firewall_1$'s child is *Core Router*, which has a different label than *Edge Router*, $Firewall_1$ is not *bisimular* to anyone.

Bisimulation Based Compression



(a) Configuration graph ℓ_c (b) Compressed configuration graph ℓ_c^{cp}

Figure 7: *Example of compression*

Algorithm 2 presents the compression algorithm on the given graphs ℓ_c, $\wp_c$ and T, where ℓ_c and T are compressed according to $\wp_c$. Before the compression, we need to first label the nodes in ℓ_c and T according to $\wp_c$: all the nodes that are presented in $\wp_c$ are labeled uniquely. Therefore, the information in the policy graphs will not be lost after compression. We then compute *bisimulation relation* on ℓ_c using the algorithms presented in [9] and then compress the graphs based on the bisimularity. However, unlike ℓ_c and $\wp_c$, T is not a directed graph, and thus the original algorithm is not applicable. To compute T^{cp}, we first compress the parts in T that overlap with ℓ_c according to the undirected version of ℓ_c^{cp}. Then we draw edges between the non-overlapping parts and the compressed parts with their original edges in T. The time complexity of the compression algorithm is $O(|E|log|V|)$. Figure 7(b) shows the compression result on graph ℓ_c. $Firewall_2$ and $Firewall_3$ are *bisimular* and are compressed to a new clustering named FW_2. $Firewall_1$ stays by itself as FW_1.

Algorithm 2 Graph pattern preserving compression

```
procedure GRAPHCOMPRESSION(ℓc, ℘c, T)
    compute the maximum bisimulation relation BR of ℓc
    compute the clusters clusters = V/BR
    collapse the nodes in the each cluster ∈ clusters
    compute compressed ℓc^cp, T^cp
    return ℓc^cp, T^cp
```

We evaluate the compression algorithm on a simulated fattree topology and a large enterprise network. We denotes the compression rate r_c as the ratio of the number of the remaining nodes in ℓ_c^{cp} to the number of the nodes in ℓ_c. From the compression result shown in Table 2, we can conclude that the compression algorithm could result in a much smaller graph for a large-scale network.

Topology	$1 - r_c$
Fattree (6750 hosts, 1125 switches)	99.38%
Enterprise (236 routers)	88.98%

Table 2: Compression results.

Incremental Compression Further leveraging the incremental compression algorithm from [10], we incrementally maintain the compressed configuration graphs. In response to changes to the original graphs, the incremental algorithm computes the new compressed graph using the changes and the compressed graph as input, independent of the original graph. That is there is no need to decompress the graph to propagate the changes.

Repair Compressed Graphs With the compression module in place, when a violation is detected, the graphs are compressed first, then passed to the optimizer. Note that one compressed edge may represent a collection of edges in the original graph. This works fine with single-path reachability type of policies, such as reachability, isolation, service chaining. However, it will break Equation 14 and 15 for link-disjoint multipath policy. Our solution is to label the predecessors of each multipath policy destination node (E.g., q for policy edge (p,q)) differently, such that they are not compressed. In addition, T^{cp} is modeled as a weighted graph, where the weight on each edge is the number of original edges that the compressed edge represents. Multipath policy constraint Equation 14 is modified as shown Equation 17, while Equation 15 remains the same because there are never multiple edges pointing to the destination node q.

$$\begin{cases} \sum_{j \in NB_{T^{cp}}(i)} (x_{i,j,p,q} * weight_{i,j}) >= m \\ \sum_{j \in NB_{T^{cp}}(i)} x_{j,i,p,q} = 0 \end{cases} \quad \text{if } i = p \tag{17}$$

Map Back The last step is to map the result back to the original graph ℓ_c. The optimization result is a set of changes (added or deleted edges) on the compressed graph ℓ_c^{cp}. To map back to ℓ_c, a changed edge (c_i, c_j) could become a set of changed edges between the cluster C_i^c and cluster C_j^c. If an edge (c_i, c_j) is supposed to be added to ℓ_c^{cp}, then on ℓ_c, for every node i in the source cluster C_i^c, there should be an edge added from i to *one* of its neighbor node j that is in the target cluster C_j^c. It does not matter which neighbor node is chosen, because all the nodes in C_j^c are equivalent with regard to the policies, which is why they are clustered as one node. In the current design, every policy node represents a physical node, and thus a policy edge represents a one-to-one connection. However, in the future, we plan to also compress the policy graphs, enabling a policy graph node representing a cluster of nodes with similar functions. This enables policy graph edges to denote various types of connection, for example, any-to-any, one-to-many. Afterwards, those computed changes will be translated into forwarding instructions, and sent to the network devices.

Policy Perseverance Finally, we prove that the Graph-Compression algorithm (Algorithm 2) preserves the equivalence between the compressed graph G_c and the original graph G with respect to the scope of policies in section 4. As loops are repaired before the graph is compressed (§5.3), the input (G) and output (G_C) of the compression algorithm are equivalent with respect to the *loop* policy. Furthermore, on a loop-free graph, the compressed graph is proven in [10] to be equivalent to the original graph for graph pattern queries. Therefore, single-path reachability policies with bounded length and waypointing constraints are equivalent on both graphs. More specifically, let us denote $Q_r(v,u)$ as the reachability query between v and u. Intuitively, for each $Q_r(v,u)$ for G, one can show by contradiction that there exists a path from v to u in G if and only if c_v can reach c_u. Similarly, the isolation policy is preserved, as ρ_{c_v,c_u} exists iff $\rho_{v,u}$ exists. Further, as reachability is preserved, for each edge $(m,n) \in G$, there exists an edge $(c_m, c_n) \in G_c$, i.e., the length of any path is the same on G and G_c.

The load balancing and multi-path policies are more complex. We can break the load balancing policy into two requirements: *pool* and *balancing*. *Pool* in this context denotes that traffic from a single source is distributed to all of a fixed pool of nodes, which is naturally preserved as a reachability requirement. *Balancing* denotes the amount of traffic distributed to each node in the pool is equal to the amount specified by the operator. Balancing is enforced by Equation 16. Since paths are not shortened by the compression algorithm, if Equation 16 holds on G_c, it also holds on G.

We prove that this conclusion also holds for multipath policy in Appendix A. Intuitively, as the predecessors of multipath destinations in are not compressed, the link-disjoint multipath criteria is preserved after compression through the bisimulation relation back propagation and flow conservation constraints.

7 Implementation

We implemented a prototype of *NEAt* in Python. *NEAt* requires no modifications to the controller or switches. The verification engine is based on prior work [19] and we use the Gurobi Optimizer [3] within our optimization engine to solve the ILP.

NEAt's pass-through mode is implemented as a proxy between the controller and switches, listening for flow modification messages. The interactive mode is implemented as an XML-RPC API, allowing it to be compatible with applications written in any language or for any controller. In particular, *NEAt* exposes a `check()` function that accepts a set of OpenFlow flow modification messages to check against the network policy. *NEAt* updates the network model with the proposed changes, verifies the model, and searches for a set of repairs if any violations are found. The application can choose to receive the repairs as a set of OpenFlow flow modification messages or as a set of edge tuples. For example, a load

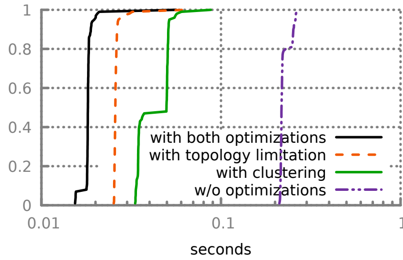


Figure 8: *Effect of optimizations*

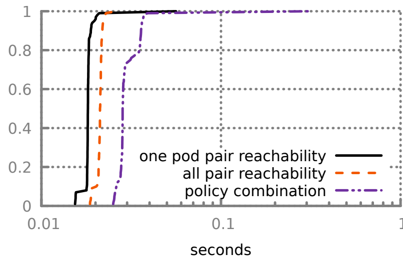


Figure 10: *Different policies.*

balancer application may wish to receive a repairs as a set of tuples (e.g., *[(s2, h1)]*) to easily re-assign a client to a particular server replica, rather than parsing an OpenFlow message from *NEAt*.

8 Evaluation

In this section, we examine the performance of repairs in *NEAt*, as well as the end-to-end latency experienced by applications. All experiments were run on a Dell Precision 5810 with a 2.6GHz Xeon E5-2697V3 CPU and 128GB RAM. We use an unmodified version of Gurobi [3] with default options in our experiments.

8.1 Repair Performance

To evaluate the feasibility and scalability of *NEAt*'s repair process, we synthesized a set of fattree topologies with various sizes, and used *NEAt* to maintain a variety of network-wide policies, including reachability, segmentation, bounded path length and multipath policies. More specifically, on each topology, under random removals of rules, we measured the repair time for each removal that caused a violation.

8.1.1 Exact matching rules

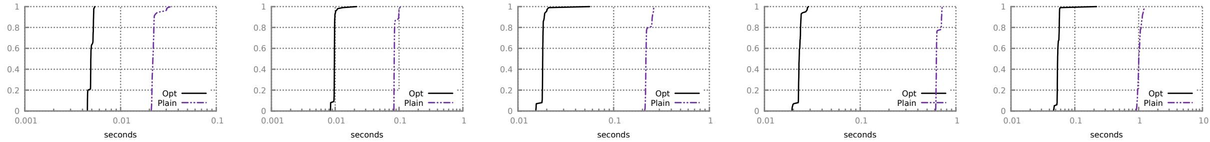
We first focus on flow-based traffic management applications, which are widely used in SDN [7, 12, 14–16]. Any forwarding rule produced by such applications at a switch matches at most one flow. In our terms, each rule only affects at most one EC.

For each fattree topology, we randomly selected a pair of pods. Suppose the desired policy is that any host in one of the pods should be able to reach every other host in both selected pods, which we will refer to as a *pod-pair reachability* policy. With random removals of rules, for those removals resulting in violations to *pod-pair reachability*, the optimization engine is triggered to perform the repair. For testing purposes, we re-verify the policy after each repair, and the check passed for all cases.

On a fattree topology with 250 hosts and 125 switches, we measured the time taken to repair *pod-pair reachability* policy by four mechanisms: (1) plain mapping, (2) mapping with topology limitation, (3) mapping with graph compression, and (4) mapping with both compression and topology limitation. Figure 8 compares the CDFs of the repair time for these four repair mechanisms: We can see the combination of graph compression and topology limitation (left most curve) brings approximately one order of magnitude speed-up over plain mapping (right most curve). Figure 9 (a-e) shows the amount of speed-up goes up as the network size scales. Even on a network with 686 hosts and 245 switches, the repair time is bounded under 0.1 second for the majority case, close to 1/20 of the repair time by plain mapping.

We next explored how *NEAt* handles a larger set of policies and a combination of different types of policies. We first assumed the desired policy being every pair of hosts should be able to reach each other , which we will refer to as an *all-pair reachability* policy. Again, on a fattree topology with 250 hosts and 125 switches, the repair time under random rule removals against this *all-pair reachability* policy was measured, as shown in Figure 10. The policy size is increased by approximately 10 times compared with *pod-pair reachability* policy, but the repair time only increases slightly.

To test a even more complex setting, next we randomly selected three pods in the fattree. Between the first two pods, hosts should be isolated from each other (*segmentation*), and between the first and third selected pods, hosts are connected by at least two path (*multipath*). For host pairs that do not fall into the previous two conditions, they are supposed to be able to reach each other (*all-pair reachability*). Both *multipath* and *all-pair reachability* are combined with a bounded path length policy, to avoid flows between pods "go through a valley". Note that unlike the previous pure single-path reachability policy, where repairs are all edge additions, in this case, a repair is sometimes a mix of edge additions and deletions. What's more, to satisfy multipath requirement, more additions are necessary. Due to this complexity, the repair time is increased, but still on the same order of magnitude of reachability policy cases, as shown in Figure 10. As verified by the re-checks, changes for fixing different types of policies keep other policy intact.



(a) 54 hosts, 45 switches (b) 128 hosts, 80 switches (c) 250 hosts, 125 switches (d) 432 hosts, 180 switches (e) 686 hosts, 245 switches

Figure 9: *Repair time comparison under random removals of exactly matching rules*

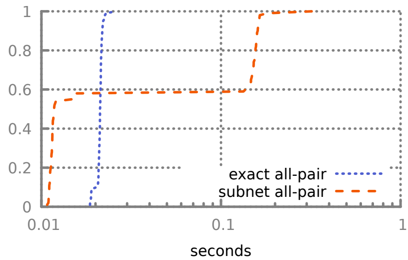


Figure 11: *Exact matching rules vs. overlapping rules*

8.1.2 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 [5, 8, 20], 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 fattree topologies. We then aggregated 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 11 compares the CDFs of the repair time for overlapping rules and exact matching rules on a 250-host-125-switch fattree topology. The repair took longer compared to applications with exact match rules because of the increased number of affected ECs. With our graph compression and topology limitation techniques, optimization is able to finish under 0.4 seconds in the worst case.

8.2 End-to-End Delay

Next, we examine the application-level delay introduced by *NEAt* when using its interactive mode. We test *NEAt* on various-sized fattree topologies using Mininet [4] and the Pox controller [6]. A learning switch application and load balancer application run on top of Pox. The load balancer balances flows between the two replicas in a round-robin fashion, and we modify it to leverage *NEAt*'s API to check the assignment of clients to replicas. If *NEAt* suggests a repair, the application updates its client-to-replica mapping with one suggested by *NEAt*. While the learning switch remains unmodified and unaware of *NEAt*, its updates are transparently checked by *NEAt*. This setup demonstrates the ability of *NEAt* to interact with the controller and applications simultaneously through its two interactive modes.

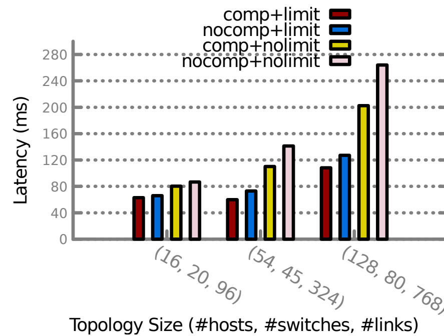


Figure 12: *Application-perceived latency of NEAt, on various fattree topologies, showing performance for a reachability policy with/without graph compression and topology limitation*

The load balancer application runs on an edge switch in the fattree topology, with clients and server replicas placed in different pods. To trigger an update, a client pings the virtual IP of the load balancer. When the appropriate event handler in the load balancer is execute, it invokes *NEAt*'s `check()` function. We measure the total latency introduced by *NEAt* as the time to invoke the `check()` function and apply it to the application's state. This includes the time to verify an update (i.e, calculate equivalence classes affected by the update, compute their configuration graphs, and verify them) and repair violations in any of the affected equivalence classes.

For each topology size, we examine the total latency for a reachability policy, with and without our compression and topology limitation optimizations. Figure 12 shows the total delay experienced by the load balancer. Topology limitation has the largest speed-up of our optimizations, but when used in combination with compression of the topology and configuration graphs, *NEAt* can verify and repair an update in under 120ms.

8.3 Enterprise Network Trace Study

Finally, we examine traces from a large enterprise network, to examine *NEAt*'s performance on real forward-

ing graphs. We examine two dumps of the data plane from 2014 and 2017. These datasets containing more than one million forwarding rules across more than 200 forwarding devices. The 2014 dataset contains 27k equivalence classes, while the 2017 trace contains 285k.

8.3.1 Bugs

For each dataset, we construct loop and reachability policies and check for violations. In the 2014 dataset, *NEAt* finds nine different loops. In the 2017 dataset, *NEAt* finds 19. We examine the forwarding table and find several of these are caused by default routes with prefix 0.0.0.0/0. Only equivalence classes with more specific rules on the device are free of loops in these cases. Another cause we discover is load balancing — a device can forward packets out one of two ports, one of which will result in a path containing a loop.

8.3.2 Synthetic Updates

Next, we use the 2017 dataset to evaluate *NEAt*'s on a data plane with a realistic number of equivalence classes. First, we repair any loops in the dataset's 285k equivalence classes. We then construct synthetic updates, choosing a destination IP address and prefix length with the same probability as they appear in the dataset's forwarding rules. An update can add a rule, delete a rule, or introduce a loop. Loops are chosen from the list of those that were discovered and repaired in the first step. An update has a 10% chance of introducing one of these loops for a particular update, which may introduce loops in multiple ECs. We generated 100 updates in this manner, which affected an average of eight ECs per update.

We apply the set of random updates to different combinations of policies, including loop-freedom, reachability, and our compression and topology limitation optimizations. Since the compression and topology limitation optimizations only apply to the reachability policy, we do not test loop freedom with compression or topology limitation. Figure 13 shows a CDF of the total update time, including verification and repairs (when necessary). Of the 100 updates, 20 loops violations needed repair, as well as 24 reachability violations. Median and 98th percentile update times were 10ms and 1300ms, respectively, for a reachability policy with compression and topology limitation enabled. For a loop freedom property, median and 98th percentile update times were 35ms and 730ms, respectively. Combining these two policies, without compression or topology limitation optimizations, resulted in median and 98th percentile times of 36ms and 193 seconds. Adding our two optimizations reduced these times to 36ms and six seconds.

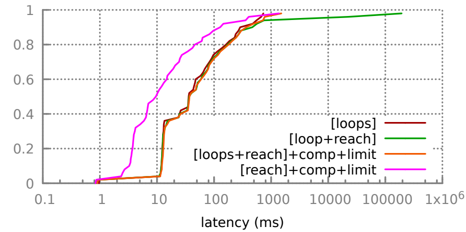


Figure 13: *Total update time for different policies and optimizations, on a model of a real-world data plane trace*

Topology Size	*NEAt*	NetGen	NetGen-C
(16, 20, 96)	5.9ms	743.2ms	513.2ms
(54, 45, 324)	7.2ms	4404.0ms	1160.8ms
(128, 80, 768)	9.0ms	16337.7ms	2056.3ms

Table 3: *Repair time of an all-pair reachability property in NEAt, NetGen [26], and NetGen using our clustering algorithm (NetGen-C) on fattree topologies. Topology size is measured as (#hosts, #switches, #edges).*

8.4 Repair vs. Synthesis

In the context of this work, we distinguish between repair and synthesis based on the scope and urgency of the change. We envision repair as a tool for temporary and immediate application on a time scale too small for human intervention. For example, the scope of *NEAt*'s repair is limited to forwarding actions after a single update to the data plane. We consider synthesis, on the other hand, to be useful for construction of longer-lived configurations and programs, without the need for a partial implementation, or changes of a larger scope.

In this section, we compare *NEAt* with NetGen [26], a tool to synthesize data plane changes using an SMT formulation. NetGen's specification language uses regular expressions to both select candidate ECs and describe changes to paths within them. In Table 3, we compare *NEAt* and NetGen under a repair scenario similar to § 8.1.1. We remove a single forwarding rule from the data plane on various-sized fattree topologies to introduce a violation, and measure the median repair time over 10 trials. For *NEAt*, we use both graph compression and topology limitation. For NetGen, we report results for the original approach operating on uncompressed configuration graphs, as well as a modified version that leverages our graph compression. We can see *NEAt* achieves performance up to two orders of magnitude faster than the modified version of NetGen, and up to four orders of magnitude faster than the unmodified version of NetGen.

In Table 4, we evaluate *NEAt* and NetGen under a synthesis scenario and generate an entire data plane from scratch. Specifically, we use *NEAt* and NetGen to "repair" an empty configuration graph and report the total time to repair an all-pair reachability policy. Since

Topology Size	*NEAt*	NetGen
(16, 20, 96)	921.7ms	7.1min
(54, 45, 324)	16.3ms	381.7min
(128, 80, 768)	2.9min	173.2hrs

Table 4: *Synthesis time of an all-pair reachability property on an empty configuration graph using fattree topologies. Topology size is measured as (#hosts, #switches, #edges).*

NEAt's policies are graph-based, we express the reachability policy with a single policy graph. As NetGen's specification is path-based, we encode the policy as a separate data plane change (i.e., regular expression) for each pair of nodes. For *NEAt*, we report the repair time with our topology limitation optimization. We do not report results with our clustering algorithm, as it is not applicable when the configuration graph is empty. We can see *NEAt* repairs an empty configuration graph more than 1000X faster than NetGen.

9 Related Work

SDN programming languages: Many programming languages have been proposed to provide abstractions to program SDNs, e.g., Frenetic [11], Pyretic [24] and Maple [29]. These allow programmers to compose complex rules without manually resolving conflicts between rules. However, these languages face limitations of expressing general policies that deliver higher-level intent, such as middleware functionality or QoS constraints.

SDN synthesis platforms: Network state can also be synthesized from a set of pre-specified correctness criteria. NetGen [26], 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 [28] and FatTire [25], this tool is designed to be used as compiler, with performance that is too slow for real-time applications (i.e., minute-scale synthesis). Instead, NEAT formulates repairs as an ILP and discovers possible repairs in under a second. While using NetGen in place of our ILP is possible, certain policies cannot be expressed in NetGen's language, such as multipath and load balancing. Similarly, Marham [13] proposes a framework for automated repair, but with performance on the order of several seconds for topologies with dozens of nodes and links. Margrave [21] analyzes changes to access control policy changes, highlighting to an operator the effect it has on the policy, without suggesting repairs to violations.

10 Limitation and Discussion

Stateful Network Applications Some stateful network applications keep internal state to provide finer grained control of network traffic. The internal state is normally constructed based on the current network state. Under pass-through mode, as *NEAt* verifies and corrects rule insertions without notifying the application, network state might become different from the application's internal state. If improperly written, the application might crash. We note that this is true for many platforms which virtualize the network [27]. This might sound unsatisfactory yet it is likely desirable, since the application may be developed by an untrusted third party, and *NEAt* can protect the network from unforeseen bugs or undesirable behavior of that application. If desired, applications could be implemented with *NEAt* in mind. We encourage developers to use interactive mode for stateful applications.

Evolving Policies In practice, the policy graph can change over time, on human time scales as network operators revise and evolve earlier policy decisions. To simplify processing, *NEAt* can pause updates while the policy graph is updated. Since loading a new policy graph is a nearly-instantaneous process, this procedure introduces minimal delay in updates reaching the network. In future work, we plan to examine how *NEAt* handles such scenarios, and make design improvements if needed.

Different Optimization Goals In the current design of *NEAt*, the repair effort uses a minimal number of edits as the optimization goal. In practice, there may be other goals, for example, ensuring critical traffic free of congestion, minimizing the amount of traffic shifts, etc. We plan to extend the design in the future to optimize user defined utility functions, and study how accurate *NEAt*'s solution is under different scenarios, and under what types of scenarios *NEAt* is applicable.

11 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* casts the repair process as an optimization problem, and repairs each update by adding or removing a minimal number of rules to satisfy the policy. Experiments on large fattree topologies show our 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. Applying *NEAt* to a large enterprise network uncovered and repaired 28 loops

We thank NSF for supporting this work with grant CNS 15-13906, and our shepherd Cole Schlesinger and the anonymous reviewers for their valuable comments.

A Multipath Policy Perseverance

Theorem 1. *(Multipath Equivalence): A multipath policy for a flow* (p,q) *holds in* G iff *the policy also holds for* (p,q) *in* G_c*.*

Proof. Consider a multipath policy that requires at least m paths for flow (p,q). Trivially, if a flow (p,q) satisfies the policy in G, the policy also holds for (p,q) in G_c, and flow conservation equations (Equation 17, 15 and 8) are satisfied.

Next, we need to prove when the policy holds in G_c, i.e., when Equation 17, 15 and 8 are satisfied, it also holds in G.

Let $path_1^c$, $path_2^c$, ... $path_n^c$ $(n \leq m)$ be the set of paths from p to q in G_c that collectively satisfy Equation 17, 15 and 8. That is, the sum of weights of all paths' starting edges is m. If n equals m, then there are at least m link-disjoint paths in G_c between p and q, and thus there are at least m link-disjoint paths in G, i.e., the policy is satisfied.

If n is less than m, then there must be at least one path in G_c, whose starting edge's weight is more than one. Let such paths be $path_{m0}^c$, ..., $path_{mj}^c$, whose starting weights are k_0, ..., k_j respectively. Consider path $path_{m0}^c$ first. The starting weight being more than one means that its starting edge is pointing from p to a cluster C_{next} which contains at least k_0 nodes which are also p's successors. Because the predecessors of q are labeled differently, each of them is a separate cluster. By the definition of bisimulation relation, two nodes are bisimular (and thus can be clustered together) only if their children's label set are the same. Via back propagation, and the constraint of Equation 8, there must be at least k_0 disjoint paths in G from p's successors in C_{next} to q's predecessors. When $path_{m0}^c$ is expanded in G, it becomes k_0 link-disjoint paths. Similarly, suppose we iterate through all the paths from p to q in G_c and expand each of them in G. As the sum of the starting weights is equal to m, there are at least m paths from p to q in G.

□

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] Mininet: Rapid prototyping for software defined networks. http://yuba.stanford.edu/foswiki/bin/view/OpenFlow/Mininet.

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

[6] The pox controller. https://github.com/noxrepo/pox.

[7] BENSON, T., ANAND, A., AKELLA, A., AND ZHANG, M. Microte: Fine grained traffic engineering for data centers. In *Proceedings of the Seventh Conference on Emerging Networking Experiments and Technologiesperiments and Technologiesnference on emerging Networking EXperiments and Technologies (CoNEXT)* (2011), ACM, p. 8.

[8] B.RAGHAVAN, M.CASADO, T.KOPONEN, S.RATNASAMY, AND A.GHODSI, A. S. S. Software-defined Internet architecture: Decoupling architecture from infrastructure. In *HotNets* (2012).

[9] DOVIER, A., PIAZZA, C., AND POLICRITI, A. A fast bisimulation algorithm. In *CAV* (2001), vol. 2102, Springer, pp. 79–90.

[10] FAN, W., LI, J., WANG, X., AND WU, Y. Query preserving graph compression. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data* (2012), ACM, pp. 157–168.

[11] FOSTER, N., HARRISON, R., FREEDMAN, M. J., MONSANTO, C., REXFORD, J., STORY, A., AND WALKER, D. Frenetic: A network programming language. In *ICFP* (2011).

[12] HELLER, B., SEETHARAMAN, S., MAHADEVAN, P., YIAKOUMIS, Y., SHARMA, P., BANERJEE, S., AND MCKEOWN, N. ElasticTree: Saving energy in data center networks. In *NSDI* (2010).

[13] HOJJAT, H., REUMMER, P., MCCLURGH, J., CERNY, P., AND FOSTER, N. Optimizing horn solvers for network repair. In *FMCAD* (2016).

[14] HONG, C.-Y., KANDULA, S., MAHAJAN, R., ZHANG, M., GILL, V., NANDURI, M., AND WATTENHOFER, R. Achieving high utilization with software-driven wan. In *SIGCOMM* (2013).

[15] JAIN, S., KUMAR, A., MANDAL, S., ONG, J., POUTIEVSKI, L., SINGH, A., VENKATA, S., WANDERER, J., ZHOU, J., ZHU, M., ZOLLA, J., HOLZLE, U., STUART, S., AND VAHDAT, A. B4: Experience with a globally-deployed software defined wan. In *SIGCOMM* (2013).

[16] JIN, X., MAHAJAN, R., LIU, H. H., GANDHI, R., KANDULA, S., ZHANG, M., REXFORD, J., AND WATTENHOFER, R. Dynamic scheduling of network updates. In *SIGCOMM* (2014).

[17] KAZEMIAN, P., CHANG, M., ZENG, H., VARGHESE, G., MCKEOWN, N., AND WHYTE, S. Real time network policy checking using header space analysis. In *NSDI* (2013).

[18] KAZEMIAN, P., VARGHESE, G., AND MCKEOWN, N. Header space analysis: Static checking for networks. In *NSDI* (2012).

[19] KHURSHID, A., ZOU, X., ZHOU, W., CAESAR, M., AND GODFREY, P. B. VeriFlow: Verifying network-wide invariants in real time. In *NSDI* (2013).

[20] M.CASADO, T.KOPONEN, S.SHENKER, AND A.TOOTOONCHIAN. Fabric: A retrospective on evolving sdn. In *HotSDN* (2012).

[21] NELSON, T., BARRATT, C., DOUGHERTY, D. J., FISLER, K., AND KRISHNAMURTHI, S. The margrave tool for firewall analysis. In *LISA* (2010).

[22] PLOTKIN, G. D., BJRNER, N., LOPES, N. P., RYBALCHENKO, A., AND VARGHESE, G. Scaling network verification using symmetry and surgery. In *POPL* (2016).

[23] PRAKASH, C., LEE, J., TURNER, Y., KANG, J.-M., AKELLA, A., BANERJEE, S., CLARK, C., MA, Y., SHARMA, P., AND ZHANG, Y. PGA: Using graphs to express and automatically reconcile network policies. In *SIGCOMM* (2015).

[24] REICH, J., MONSANTO, C., FOSTER, N., REXFORD, J., AND WALKER, D. Modular sdn programming with pyretic. In *USENIX ;login, 38(5)* (October 2013), pp. 40–47.

[25] REITBLATT, M., CANINI, M., GUHA, A., AND FOSTER, N. Fattire: Declarative fault tolerance for software-defined networks. In *HotSDN* (2013).

[26] SAHA, S., PRABHU, S., AND MADHUSUDAN, P. NetGen: Synthesizing data-plane configurations for network policies. In *SOSR* (2015).

[27] SHERWOOD, R., GIBB, G., YAP, K.-K., APPENZELLER, G., CASADO, M., MCKEOWN, N., AND PARULKAR, G. Can the production network be the testbed? In *OSDI* (2010).

[28] SOULE, R., BASU, S., MARANDI, P. J., PEDONE, F., KLEINBERG, R., SIRER, E. G., AND FOSTER, N. Merlin: A language for provisioning network resources. In *CoNEXT* (2014).

[29] VOELLMY, A., WANG, J., YANG, Y. R., FORD, B., AND HUDAK, P. Maple: Simplifying sdn programming using algorithmic policies. In *SIGCOMM* (2013).

[30] ZHOU, W., JIN, D., CROFT, J., CAESAR, M., AND GODFREY, P. B. Enforcing customizable consistency properties in software-defined networks. In *NSDI* (2015).