Safely and Automatically Updating In-Network ACL Configurations with Intent Language

Bingchuan Tian[§⋆†], Xinyi Zhang[◇⋆†], Ennan Zhai[⋆], Hongqiang Harry Liu[⋆], Qiaobo Ye[⋆], Chunsheng Wang[⋆], Xin Wu[⋆], Zhiming Ji[⋆], Yihong Sang[⋆], Ming Zhang[⋆], Da Yu[•⋆], Chen Tian[§], Haitao Zheng[△], Ben Y. Zhao[△]

Nanjing University[§], University of California Santa Barbara[◇], Brown University[•], University of Chicago[△]
Alibaba Group[⋆]

ABSTRACT

In-network Access Control List (ACL) is an important technique in ensuring network-wide connectivity and security. As cloud-scale WANs today constantly evolve in size and complexity, in-network ACL rules are becoming increasingly more complex. This presents a great challenge to the updating process of ACL configurations: network operators are frequently required to update "tangled" ACL rules across thousands of devices to meet diverse business requirements, and even a single ACL misconfiguration may lead to network disruptions. Such increasing challenges call for an automated system to improve the efficiency and correctness of ACL updates. This paper presents Jinjing, a system that aids Alibaba's network operators in automatically and correctly updating ACL configurations in Alibaba's global WAN. Jinjing allows the operators to express in a declarative language, named LAI, their update intent (*e.g.*, ACL migration and traffic control). Then, Jinjing automatically synthesizes ACL update plans that satisfy their intent. At the heart of Jinjing, we develop a set of novel verification and synthesis techniques to rigorously guarantee the correctness of update plans. In Alibaba, our operators have used Jinjing to efficiently update their ACLs and have thus prevented significant service downtime.

CCS CONCEPTS

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

KEYWORDS

Access Control List; Domain Specific Language; Network Configurations; Verification; Synthesis

ACM Reference Format:
Bingchuan Tian, Xinyi Zhang, Ennan Zhai, Hongqiang Harry Liu, Qiaobo Ye, Chunsheng Wang, Xin Wu, Zhiming Ji, Yihong Sang, Ming Zhang, Da Yu, Chen Tian, Haitao Zheng, Ben Y. Zhao. 2019. Safely and Automatically Updating In-Network ACL Configurations with Intent Language. In *SIGCOMM '19: 2019 Conference of the ACM Special Interest Group on Data Communication, August 19–23, 2019, Beijing, China.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3341302.3342088

[†]Both authors contributed equally to the paper.

SIGCOMM '19

ACM ISBN 978-1-4503-5956-6/19/08…$15.00
https://doi.org/10.1145/3341302.3342088

1 INTRODUCTION

In-network Access Control Lists (ACLs), which are configured in network devices, play a critical role in ensuring network-wide connectivity, security, and reliability. Compared with other ACLs, such as end-host firewalls, in-network ACLs filter unwanted traffic early on by offering high-throughput, distributed traffic control in the backbone and edge networks, effectively protecting core services. Current global service providers widely employ in-network ACL configurations in the management of their networks.

In recent years, in-network ACL rules in the modern Wide Area Networks (WANs) are growing increasingly complex due to the following three reasons: (1) cloud-scale WANs today are constantly evolving in size and complexity; (2) their hosted services are also becoming increasingly sophisticated, creating complicated network dependencies; and (3) in-network ACL rules across thousands of devices are tangled with numerous routing paths, giving rise to very complex logical relations.

As a result, such complexity of ACL rules presents significant challenges to *the process of updating ACL configurations: updating ACL configurations in the WAN— a frequently needed network operation—is error-prone, and even a single ACL misconfiguration may lead to service disruptions.*

Alibaba is one of the largest global service providers, and is offering tens of diverse global services. We are also faced with the above-mentioned ACL update challenges in our global-WAN management. For example, as new services are constantly deployed in our WAN, our network operators are frequently asked to update the ACL rules to permit new IP prefixes; however, updates like these often produce unexpected side effects elsewhere. As another example, in one of our recent WAN upgrades, our operators were asked to migrate ACL rules from a layer of core routers to a group of specific gateways with the aim to reassign the core routers as the provider-edge (PE) routers, while preserving traffic reachability. Since the migration involves the reconfiguration of 30% of all routers in our global WAN, each of which managing thousands of routing paths, it took operators multiple weeks to design the migration and roll-back plans, to discuss and assess the plans, and finally to execute the operation. Despite efforts such as these, ACL updates still run the risk of containing errors that could cascade into worst-case scenarios for the operator.

This stark reality calls for an automated system to increase the efficiency and correctness of our network operators' work, one that can advise our operators to automatically update ACL configurations based on their intent.

1.1 Our Approach: Jinjing

This paper presents Jinjing, a system that automatically generates ACL update plans based on our operators' high-level intent. Jinjing offers an intent language, named LAI, with natural primitives, which our operators can use to express their objectives in ACL configuration update. Then, Jinjing parses the LAI-program and automatically generates update plans satisfying the expressed intent.

An LAI-program should contain three parts: region, requirement, and command. Region specifies an update scope and what devices are allowed to update. Requirement describes the *desired reachability* for an update. For example, requirements can mean 1) to preserve reachability during ACL migration, or 2) to permit a new prefix in the updated ACLs, while keeping all other reachability the same. Command defines specific operations to be performed under the constraints specified by region and requirement. After surveying our network operators, three operation primitives with increasing degrees of automation are designed: **check**, **fix** and **generate**. Figure 3 shows an example LAI program.

The design of the three operations—**check**, **fix** and **generate**—presents their own challenges.

Primitive: check. This primitive aims to check whether an updated ACL configuration achieves the desired reachability, meaning it satisfies the reachability specified in the intents while preserving the original properties (*i.e.*, without side effects). This is challenging because we not only need to check whether the desired reachability is satisfied, but also whether there are any side effects. Unfortunately, state-of-the-art solutions are insufficient. Specifically, existing configuration verification techniques (*e.g.*, firewall and router configuration verification [12, 13, 15, 23, 28, 34]) check the reachability of a given individual prefix or a pair of routers (*i.e.*, "can A talk to B?"), rather than whether all prefixes or pairs of routers achieve the desired reachability within one-run. In other words, using the state-of-the-art verification tools, we have to enumerate all prefixes and all paths to check their reachability of the two network snapshots before and after the update. This is prohibitively expensive. While a recent verification tool, Minesweeper [1], can check the desired reachability of multiple packets and paths within one-run, Minesweeper scales to hundred devices—far short of the scale of WAN [2]; meanwhile, due to the asymmetry of our WAN, compression techniques [2] cannot be used to speed up the state-of-the-art verification tools in our WAN.

We address this challenge in two steps. First, we propose a novel theorem (Theorem 4.1) to help us identify a small set of ACL rules affected by the update. Then, we only need to encode a small "delta " formula rather than a big formula, like Minesweeper, representing the entire ACL configurations across many routers. Solving such a small formula is very efficient.

Primitive: fix. For any update that fails to achieve the desired reachability, the **fix** primitive generates a fixing plan that repairs the violation. Similar to the checking case, existing control plane repair efforts, like CPR [14], can only repair the reachability violation of single prefix or packet, rather than fixing all violations in an update.

We address this challenge with the help of **check**. We first use **check** to efficiently get counter examples, then group all counter examples with the same path decision into several equivalence classes, represented in ACL rule format. To decide where to place rule-formed fixes for these classes, we build a formula that models the desired reachability, and then combine the formula with constraints that describe where to place each fixing rule. Solving this combined formula tells us how the fixing rules should be added.

Primitive: generate. This primitive generates ACL configurations from scratch. Related work falls into configuration synthesis area. Specifically, the state-of-the-art configuration synthesis tools (*e.g.*, Propane [3], PropaneAT [4], SyNET [10] and NetComplete [11]) offer some inspirations. However, these tools are focused on generating configurations for specific network protocols such as BGP and OSPF. Because their synthesis algorithms heavily rely on protocol-specific features (*e.g.*, BGP update propagation), they cannot be easily extended to synthesizing ACL-level configurations. Although SyNET [10] is able to synthesize general in-network configurations, SyNET scales poorly to large networks [11].

To generate ACLs from scratch in a scalable way, we need to find for each device a suitable ACL decision function for all packets (*i.e.*, permit or deny) in order to achieve the desired reachability. However, this is hard because finding such decision functions requires us to enumerate all packets and all devices, which is quite time-consuming. Even with the decision functions, the second challenge of translating them into ACLs is non-trivial. We tackle both of the above challenges by grouping packets based on the ACL equivalence class, thus greatly reducing the search space. More importantly, the decision functions on ACL equivalence classes can be naturally translated to well-formed ACL rules, solving our second challenge. We further propose optimization techniques to improve scalability.

Real-world deployment. Alibaba's network operators have used Jinjing in managing their ACL update in our WAN. Jinjing successfully prevents service downtime in their operations. Furthermore, Jinjing is efficient in generating large-scale ACL update plans, taking less than 15 minutes in our global network. We discuss our experience with Jinjing in §7 and §8.

2 BACKGROUND AND MOTIVATION

This section first describes what is in-network access control list in §2.1, and then presents the motivation in §2.2.

2.1 Background: ACL

An access control list (ACL) is a sequential collection of permit and deny conditions, called ACL rules. Packets to be permitted (or denied) are identified in terms of their source/destination IP addresses and ports, and protocol, *i.e.*, a 5-tuple like ⟨*sip*, *dip*, *sport*, *dport*, *proto*⟩.

Packet that comes into a router is compared to ACL rules from top to bottom until it has a match. If no matches are found before reaching the end of the list, the packet is either denied or permitted by the default rule that matches all packets. For example, in Figure 1, all the ACLs employ "permit all" as the default rule.

ACLs can be applied to both ingress and egress interfaces of a router. They specify what traffic is allowed in or out of the device. Such a feature should be explicitly specified by the operators when configuring routers.

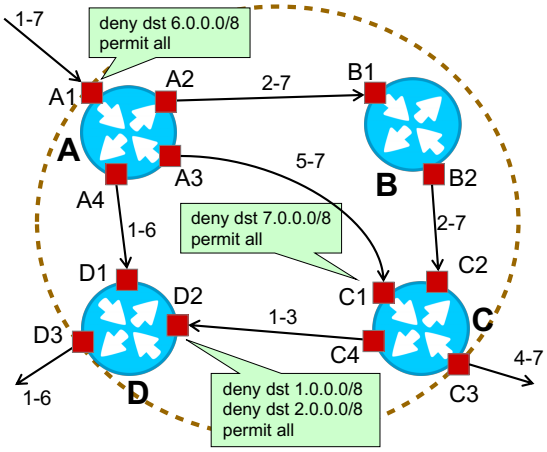


Figure 1: Motivating example. X_i means the interface i of router X. In this example, interfaces A_1, C_1 and D_2 have ACL rules. The number on each edge represents the routing directions of the corresponding traffic; *e.g.*, "1" represents the traffic towards 1.0.0.0/8, "2" for 2.0.0.0/8, *etc.*

2.2 Motivation and Our Goal

Taking Figure 1 as an example, there are four routers in this small sub-network. A_i means the interface i of router A. The number on each edge represents the forwarding directions for different types of traffic in the network. For example, "1-6" on the edge from A_4 to D_1 means all traffic with destination "1.0.0.0/8" to "6.0.0.0/8" can be routed to D_1. Even for such a small network, ACL misconfiguration can easily happen. Suppose the operator wishes to clean up all ACL rules on device D, while preserving the packet reachability passing through this subnet. Since D_2 no longer holds the ACL to deny traffic to 1.0.0.0/8 and 2.0.0.0/8, some other interfaces need to have new ACL rules to serve this functionality. The operator may decide to follow the common practice and deny such traffic on the gateway interface A_1. What she failed to realize is that, by moving the two deny rules from D_2 to A_1, the traffic going to these prefixes can no longer go through the path $\langle A1, A4, D1, D3\rangle$, violating the desired packet reachability—Traffic 1 and 2 can originally exit the subnet from D_3, but it fails after the update. Although errors like this can be avoided by carefully planning ACL updates and checking through all affected paths, thorough checking quickly gets impractical as the network increase in scale.

Motivating scenarios for ACL update tasks. We surveyed our network operators about typical ACL update tasks, and list the most common (but not limited to) tasks:

- **Opening/isolating traffic:** Service deployments, retirements and upgrades happen on a daily basis. Our operators are frequently asked to allow or isolate certain traffic by modifying ACLs, while preventing any side effect.
- **Checking ACL updates:** A typical ACL update requires our operators to manually come up with an ACL update plan and is, not surprisingly, error-prone. Our operators are in dire need of a means to check whether the updated ACL configurations violate the desired reachability.

prog	::=	*region*; *req*; *cmd*	LAI program
region	::=	**scope** *l* ⟨*n*⟩; **allow** *l* ⟨*n*⟩	Defining scope and interfaces allowed to modify ACL rules
req	::=	**modify** *l* ⟨*n*⟩ **to** *l* ⟨*n′*⟩	Specifying ACL updates
	|	**control** *n* ⟶ *n* (**isolate**|**open**|**maintain**) *h*	Specifying desired reachability
cmd	::=	**check**	Checking desired reachability
	|	**fix**	Generating fixing plan
	|	**generate**	Generating new ACLs
l ⟨*n*⟩	::=	*nil* | *n* **and** *l*	Interface list
n	::=	*device* : *interface*	A set of devices holding ACLs
n′	::=	*device′* : *interface′*	ACL-updated interfaces
h	::=	**src** *prefix* | **dst** *prefix*	Specifying src or dst prefix

Figure 2: Intent language LAI's grammar.

- **Correcting buggy ACL updates:** The operators want to fix buggy update plans, which fail to achieve the desired reachability, without any side effect.
- **ACL migration:** Major changes like network upgrades require operators to move ACLs from one group of devices to another specified group, while maintaining the original reachability.

Any of the above network-wide ACL update operations, if not done properly, could result in significant service downtime.

Goal. Our goal is to build a system that aids our operators to automatically handle their ACL updates. The operators only need to provide their intent, *i.e.*, the end requirements of ACL update tasks, and then our system automatically generates a plan that meets the operators' intent in an efficient way. Such a system would significantly simplify the ACL update process, ensure the correctness of ACL configurations, and minimize the burden placed on our network operators.

3 JINJING OVERVIEW

JINJING takes in a high-level ACL update intent, along with the current network configuration and topology. Then, JINJING generates the update plan satisfying the intent. In this section, we detail the design of intent language, named LAI (§3.1), with a running example to show how an operator expresses an ACL update task in LAI (§3.2). Finally, we present our system model used in designing the primitives of LAI (§3.3).

3.1 Intent Language: LAI

The operator is allowed to express diverse ACL update tasks with an intent language, LAI (or Language for ACL Intents). Figure 2 shows LAI's grammar, which includes global variable *node*, list *l* ⟨*node*⟩ and primitives.

To write an LAI program, the operator needs to specify three parts: `region`, `req`, and `cmd`, as shown in the first line of Figure 2. The operator can express diverse ACL update tasks by combining these commands (as shown in Table 1).

Region (`region`). To express an ACL update task, the operator should specify a management scope, Ω, using **scope**. Typically, a management scope could be a cluster, a layer of routers, an availability zone, or even the entire network. For devices out of this scope, ACLs within the scope can be regarded as a black box, meaning that traffic permitted on each link can be arbitrarily changed as long as it has no unintended effect on traffic going through the scope. For example, in Figure 1, the dashed circle is the specified

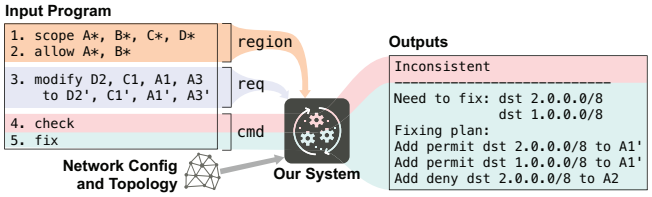


Figure 3: Our running example. Our system takes the input program and network topology to generate update plans. "A*" means all interfaces in A.

scope, and the operator wants to make sure traffic going from A_1 to C_3 and D_3 has the desired reachability. In Jinjing's usage, the scope Ω totally depends on ACL update tasks in hand. **allow** defines a set of interfaces where we are allowed to update ACL rules;

Requirement (`req`) offers the operator two ways to express her update requirements: 1) updating ACL configuration and 2) updating packet reachability. **modify** specifies ACL configuration updates to be examined (*i.e.*, $I\langle n\rangle$ and $I\langle n'\rangle$); **control** specifies the desired packet reachability update, *i.e.*, what traffic to isolate, open or maintain. If **control** is not explicitly specified in an LAI program, the original reachability passing through the given scope should be maintained.

Command (`cmd`) specifies the update operations under the given requirements. Our system can **check** if the specified ACL updates achieve the desired packet reachability, **fix** the given ACLs by adding rules on top, or **generate** new ACLs to achieve the desired packet reachability. The abovementioned primitives offer the operators a flexible ACL-update way to either manually write ACL configuration updates and then check/fix, or automatically generate ACLs from scratch.

In production networks, not all ACL updates are suitable to be automatically generated due to the reasons such as maintainability and operability. For example, when the operators try to manually locate the root cause for some network failure, unreadable but automatically-generated ACL rules may significantly affect the diagnosis progress. As a result, over-automation may create the potential barriers for the maintainability and operability of ACL configurations. We, therefore, offer different primitives (*e.g.*, `check`, `fix`, and `generate`) with increasing degrees of automation, rather than dropping all update tasks to a single `generate` primitive.

Summary. Table 1 shows how to combine different primitives to write LAI programs for common ACL update tasks.

3.2 A Running Example

Suppose in Figure 1, the operator wants to "clean up" the ACLs on device C and D, so that she moves "deny dst 1.0.0.0/8, deny dst 2.0.0.0/8" from D_2 to the top of A_1's ACL, forming an updated ACL denoted as A_1', which includes a total of four rules: "deny dst 1.0.0.0/8, deny dst 2.0.0.0/8, deny dst 6.0.0.0/8, permit all." And she also moves "deny dst 7.0.0.0/8" from C_1 to interface A_3 to form A_3'. With Jinjing, the operator writes the LAI program as shown in Figure 3.

The program specifies **scope** involves all the interfaces of A, B, C, and D (*i.e.*, the dashed circle in Figure 1). Since she wishes to clean up C and D, only interfaces on A and B are **allow**ed. The **modify** involves setting D_2 and C_1 to D_2' and C_1' which permit all traffic, while changing A_1 and A_3 to include the additional deny rules. Note that since no **control** primitive is explicitly used, the desired reachability is to keep original packet reachability. After specifying the update requirement, she then **check**s whether the update achieves the desired reachability, in this case, the original traffic reachability. As the system outputs "inconsistent" after checking, she goes on to **fix** the update. Our system returns a fixing plan as shown in Figure 3, which adds the rules "permit dst 2.0.0.0/8" and "permit dst 1.0.0.0/8" to A_1'. Note that the fixed ACLs may contain redundant rules which can be further simplified.

Roadmap. The example above belongs illustrate how Jinjing handles the checking and fixing of ACL update plans. Table 1 summarizes the LAI primitives used in this example in row one, and also lists how other common ACL update tasks (listed in §2.2) can be expressed using an LAI program.

Making primitives (*e.g.*, **check**, **fix**, and **generate**) functionally correct and efficient is challenging, because they entail verification or synthesis tasks in a large network. In §4, §5 and §6, we use the three ACL update tasks in Table 1 as examples, respectively, and detail the designs of the highlighted primitives.

3.3 System Model

Before elaborating on the primitives' designs, we introduce some basic concepts and definitions used throughout the paper, with Table 2 defining the basic symbols.

ACL and ACL decision model. We use L_ξ to denote the ACL applied to interface ξ. For example, in Figure 1, L_{A_1} denotes an ACL applied to interface A_1 consisting of two ACL rules: "deny dst 6.0.0.0/8" and "permit all." $f_\xi(h)$ is used to denote a boolean function: given a packet h, whether the decision of L_ξ on h is permit (return TRUE) or deny (return FALSE). We call f_ξ the decision model of ACL L_ξ.

Border interface. For an interface ξ of some device in the given subnet scope Ω, if ξ receives/sends traffic from/to outside Ω, ξ is called a border interface. In Figure 1, interfaces A_1, C_3, and D_3 are border interfaces.

Path and path decision model. Given a pair of border interfaces (*e.g.* s and d) with respect to Ω, a path p is a list of device interfaces from s to d. Note that a path's source and destination must both be border interfaces. Consider the example in Figure 1. There are three paths from $A1$ to $D3$: $p_0 = \langle A_1, A_4, D_1, D_3\rangle$, $p_1 = \langle A_1, A_3, C_1, C_4, D_2, D_3\rangle$ and $p_2 = \langle A_1, A_2, B_1, B_2, C_2, C_4, D_2, D_3\rangle$.

We define a path p's path decision model, c_p, as: the conjunction of ACL decision models of all the interfaces on p. Formally, a path

Table 1: Primitives for common update tasks. Highlighted primitives detailed in corresponding sections.

ACL Update Tasks	Primitives Used	Sec.
ACL update plan checking and fixing	scope, allow, modify, **check**, **fix**	§4
ACL migration	scope, allow, modify, **generate**	§5
Opening/isolating traffic for service	scope, allow, **control**, generate	§6

Table 2: Important Symbols

Symbol	Type	Description
h	bool vec	a packet header (or packet)
L	list	an ACL containing a list of ACL rules
L_ξ	list	the ACL applied to interface ξ
$f_\xi(h)$	bool func	the decision model of the ACL L_ξ on a packet h, *i.e.*, permit or deny
$\mathcal{L}_\Omega$	set	the group containing the ACLs of all the devices in a specified subnet Ω
$\mathcal{F}_\Omega$	set	the set of decision models of ACLs in $\mathcal{L}_\Omega$
$\mathcal{P}_\Omega$	set	all the paths in the subnet Ω
$c_p(h)$	bool func	the aggregated decision model (path decision model) of all ACLs on path p
$\partial_{i,j}(h)$	bool func	the forwarding model from interface i to j return *TRUE* or *FALSE*
$\mathcal{G}_\Omega$	set	the set of all forwarding models in the subset Ω
$m_j(h)$	bool func	whether rule j matches h

p's path decision model c is:

$$c_p \triangleq \bigwedge_{i \in p} f_i \tag{1}$$

As shown above, a path's decision model is a boolean function that tells whether an input packet h is permitted or denied by this path. For instance, p_0's decision model is: $c_{p_0} = f_{A_1} \wedge f_{A_4} \wedge f_{D_1} \wedge f_{D_3}$.

ACL group and update. For a given subnet Ω, the ACL group for this subnet, $\mathcal{L}_\Omega$, contains the ACLs of all interfaces in Ω. In Figure 1, $\mathcal{L}_\Omega = \{L_{A_1}, L_{C_1}, L_{D_2}\}$. We use $\mathcal{L}_\Omega$ and $\mathcal{L}'_\Omega$ to represent the ACL groups before and after update, in Ω, respectively. A decision model group $\mathcal{F}_\Omega$ is defined as a set containing the ACL decision models of all interfaces in Ω.

Packet reachability consistency. When **control** is not specified, our update needs to achieve consistency with the original reachability, defined as packet reachability consistency. In the given subnet Ω, packet reachability consistency between two ACL groups, $\mathcal{L}_\Omega$ and $\mathcal{L}'_\Omega$, is achieved, if and only if each path p's decisions on all traffic through p do not change if we update $\mathcal{L}_\Omega$ with $\mathcal{L}'_\Omega$.

Desired reachability consistency. When **control** is specified, we adopt a modified definition of consistency. In the subnet Ω, desired reachability consistency is achieved, if and only if, for each path p, its decision model based on $\mathcal{L}'_\Omega$ is equivalent to the desired decision model, which is derived from $\mathcal{L}_\Omega$ and updated based on the **control** specifications.

For better clarity, we focus on packet reachability consistency in §4 and §5 and discuss how our designs can be extended to support desired reachability consistency in §6.

4 ACL UPDATE CHECKING & FIXING

We now start by explaining how the **check** primitive verifies manually written ACL configuration updates (§4.1), and when an update fails the check, how **fix** is performed to patch it up (§4.2), using the running example from §3.2.

In our example first introduced in §3.2, since **control** is not specified in the requirement part, **check** and **fix** primitives check and fix the packet reachability consistency, respectively—*i.e.*, whether the reachability preserves before and after the ACL update. We describe a trivial way to extend **check** and **fix** for the desired reachability case in §6.

4.1 Primitive Design: check

check takes as inputs: 1) a scope Ω specified by the **scope** requirement, and 2) ACL groups before and after the update, $\mathcal{L}_\Omega, \mathcal{L}'_\Omega$, which are specified by the **update** requirement.

In addition, **check** extracts all traffic (*e.g.*, Traffic 1-7 in the Figure 1) entering Ω from our internal IP management system. Cloud providers today maintain their own IP management systems used to store data like IP prefixes and routing tables, and extracting traffic based on this IP information is trivial in practice.

Forwarding equivalence class. We use boolean functions $\partial_{i,j}(h)$ to model the forwarding behavior. Specifically, $\partial_{i,j}(h)$ represents whether interface i forwards a packet h to interface j. We use $\mathcal{G}_\Omega$ to denote the set of all $\partial_{i,j}$, where $\langle i, j\rangle$ is a directed link in the subnet Ω. Given two packets h_x and h_y, if Equation (2) returns *TRUE*, we say h_x and h_y belong to the same forwarding equivalence class.

$$\bigwedge_{\partial \in \mathcal{G}} (\partial(h_x) = \partial(h_y)), \tag{2}$$

We use $[h]_{FEC}$ to denote the forwarding equivalence class (FEC) exemplified by h. For example, there are five FECs in our example: $[1]_{FEC} = \{1\}$, $[2]_{FEC} = \{2, 3\}$, $[4]_{FEC} = \{4\}$, $[5]_{FEC} = \{5, 6\}$ and $[7]_{FEC} = \{7\}$.

We also define $\psi_{[h]_{FEC}}(h_i)$ as a boolean function, which indicates whether a packet h_i belongs to the FEC $[h]_{FEC}$.

Basic version. The intuition of **check**'s basic design is to first classify all traffic entering the given scope into forwarding equivalence classes, and then use an SMT solver to check the packet reachability consistency for each forwarding equivalence class. Algorithm 1 details the design.

Initially (line 1), we encode $\mathcal{L}_\Omega$ and $\mathcal{L}'_\Omega$ to their decision model sets $\mathcal{F}_\Omega$ and $\mathcal{F}'_\Omega$, respectively. Then, we find all paths in Ω, putting them in $\mathcal{P}$ (line 2)[1]. Next, we get E which contains the FECs of all traffic entering Ω (line 3-5).

For each FEC, $[h_i]_{FEC}$ in E, we take the following two steps:

- *Step 1* (line 8-10): We iterate over all paths p in Ω, and check if p_i allows $[h_i]_{FEC}$ to go through Ω, *i.e.*, entering s and successfully leaving from d. If so, we put p_i in a set $\mathcal{Y}$.
- *Step 2* (line 11-13): With $\mathcal{F}_\Omega$, $\mathcal{F}'_\Omega$, and $\mathcal{P}$, we can get c_p and c'_p by Equation (1). Consistency for $[h_i]_{FEC}$ is achieved when and only when $\mathcal{L}_\Omega$ and $\mathcal{L}'_\Omega$ behave the same on all paths in $\mathcal{Y}$. Thus, we use SMT solver (*e.g.*, Z3) to check the packet reachability consistency of $[h_i]_{FEC}$ by the equation:

$$\neg \left(\bigwedge_{p \in \mathcal{Y}} c_p \Leftrightarrow c'_p \right) \wedge \psi_{[h_i]_{FEC}} \tag{3}$$

If any of the above verifications are satisfiable, that means $\mathcal{L}_\Omega$ and $\mathcal{L}'_\Omega$ are making different decisions for some packets on certain paths—in this case, the operator launches a **fix**ing process (§4.2); otherwise, $\mathcal{L}'$ maintains packet reachability consistency.

In theory, classifying equivalence classes may produce explosive growth in number of FECs. For example, n randomly generated

[1]Cloud-scale networks are typically structured and the topology information is well managed, so that paths are enumerable in polynomial time, from the perspective of routing DAGs.

Algorithm 1: Basic version: Packet reachability checking

Input: $\mathcal{L}_\Omega$, $\mathcal{L}'_\Omega$: ACL groups before and after update.
Input: X_Ω: The set of traffic entering Ω.
Output: `Consistent`, or `Inconsistent`.

$\mathcal{F}_\Omega \leftarrow \mathcal{L}_\Omega$, $\mathcal{F}'_\Omega \leftarrow \mathcal{L}'_\Omega$
$\mathcal{P} \leftarrow$ find all p in Ω
$E \leftarrow \emptyset$
foreach $x \in X_\Omega$ **do**
  $E.append([x]_{FEC})$
foreach $[h_i]_{FEC} \in E$ **do**
  $\mathcal{Y} \leftarrow \emptyset$
  foreach $p_i \in \mathcal{P}$ **do**
    if p_i *allows packets in* $[h_i]_{FEC}$ *to pass through* Ω **then**
      $\mathcal{Y}.append(p_i)$
  $r \leftarrow$ solve $\bigwedge_{p \in \mathcal{Y}} \neg\, c_p \Leftrightarrow c'_p \wedge \psi_{[h_i]_{FEC}}$ via SMT solver
  if r *is satisfiable* **then**
    return `Inconsistent`
return `Consistent`

routing rules may lead to unavoidable 2^n equivalence classes in the worst case. However, in a well-organized cloud-scale network where traffic is grouped and converged, we have never observed the equivalence class explosion in practice. We discuss this more in §9.

Example. In the §3.1 example, when we look at $[2]_{FEC} = \{2, 3\}$, which enters Ω at interface A_1 and leaves from D_3. There are two paths p_0 and p_2 for $[2]_{FEC}$: $p_0 = \langle A_1, A_4, D_1, D_3\rangle$, and $p_2 = \langle A_1, A_2, B_1, B_2, C_2, C_4, D_2, D_3\rangle$. Thus, $\mathcal{Y} = \{p_0, p_2\}$, and Equation (3) takes the form of $(\neg(c_{p_0} \Leftrightarrow c'_{p_0}) \vee \neg(c_{p_2} \Leftrightarrow c'_{p_2})) \wedge \psi_{[2]_{FEC}}$. Since "deny dst 2.0.0.0/8" is moved from D_2 to A_1, the reachability for traffic going to 2.0.0.0/8 (*i.e.*, Traffic 2) on p_0 is changed from $c_{p_0} = TRUE$ to $c'_{p_0} = FALSE$, making Traffic 2 a valid solution for Equation (3). This indicates that the current update plan violates the consistency of Traffic 2.

Speeding up basic Algorithm 1 via differential rules. Because many ACL updates only change a small set of rules, it is a waste to encode the entire ACL into a decision model for Equation (1), especially since solving such a huge formula via SMT solver is time-consuming. We therefore optimize Algorithm 1 by defining two important terms. One identifies the set of rules that are directly removed or added, and the other filters out rules that have no overlap with these rules.

Definition 4.1. ***Differential ACL rules.*** *Given two ACLs L and L', the differential ACL rules between L and L' is*

$$D_{L,L'} \;,\; L - L\overrightarrow{\cap}L' \;\cup\; L' - L\overrightarrow{\cap}L' \;; \tag{4}$$

where $\overrightarrow{\cap}$ is the Longest Common Sequence (LCS) of L and L'.

Definition 4.2. ***Related Rules.*** *Consider an ACL L. For a set of ACL rules S, its related rules in L are those that overlap with at least one rule in S:*

$$R(L, S) \;,\; \{k \in L : \exists k' \in S(m_k \wedge m_{k'} \text{ is satisfiable})\}; \tag{5}$$

Here $m_k(h)$ is a function that tells if a packet h can be matched by rule k. Only when rule k and k' overlap, can $m_k \wedge m_{k'}$ be *TRUE*.

Theorem 4.1. *Consider two ACLs L and L'. L is consistent to L' if $R(L, D_{L,L'})$ is consistent to $R(L', D_{L,L'})$.*

Proof. Denote H as the set of packets that can be matched by differential rules, *i.e.*, $H = \{h : \bigvee_{i \in D_{L,L'}} (m_i(h) \vee m'_i(h)) \text{ is satisfiable}\}$, where $m'_i(h)$ is a boolean function indicating whether rule i in L' matches h. Considering a packet header h, there are two possibilities. If $h \in H$, all possible rules it can match belong to either $R(L, D_{L,L'})$ or $R(L', D_{L',L})$, so that the equivalence of $R(L, D_{L,L'})$ and $R(L', D_{L',L})$ implies the same action of L and L' on h. If $h \notin H$, all possible rule it can match in L and L' are same, and the action is therefore also the same. □

Following the spirit of Theorem 4.1, we optimize Algorithm 1 by preprocessing inputs $\mathcal{L}_\Omega$ and $\mathcal{L}'_\Omega$, and filtering out unrelated rules, thus "feeding" smaller ACLs to Algorithm 1. Specifically, we first obtain the sets of rules that are added or removed, by getting $D_{L,L'}$ and $D_{L',L}$ for all (L, L') pairs in $\mathcal{L}_\Omega$ and $\mathcal{L}'_\Omega$. Second, taking the union over all the differential rules gives us a set $Diff_\Omega$, which contains all rules added or removed by the update in Ω. Next, we use $Diff_\Omega$ to simplify each L_i in $\mathcal{L}_\Omega$ based on Definition 4.2, and put the result $R(L_i, Diff_\Omega)$ in a set named $\mathcal{R}_\mathcal{L}$; similarly, we can also get another set $\mathcal{R}_{\mathcal{L}'}$ containing all the simplified L' (in $\mathcal{L}'_\Omega$). Finally, we replace Algorithm 1's inputs $\mathcal{L}_\Omega$ and $\mathcal{L}'_\Omega$ with $\mathcal{R}_\mathcal{L}$ and $\mathcal{R}_{\mathcal{L}'}$.

Using differential rules significantly reduces the size of c_p and c'_p, because the inputs only contain differential and related rules. Thus we can solve Equation (3) much faster.

ACL decision model optimization. The prior decision model encodes ACL rules sequentially according to their priority, resulting in $O(n)$ search depth in the SMT solver [39]. Inspired by tournament sort algorithm, we employ a dependency tree structure to replace the previous sequential encoding. This optimization reduces the average search depth to $O(\log n)$.

4.2 Primitive Design: fix

Once inconsistency is detected, the operator can use the **fix** primitive to correct the inconsistency. This consists of two phases: seeking neighborhoods and generating a fixing plan.

Seeking neighborhoods. If there is an inconsistency, the SMT solver can only return us one solution, *e.g.* h, which is just one of the root causes causing the ACL to be inconsistent. To get all the root causes, we first "*enlarge*" the obtained solution to cover a set of packets resulting in the same inconsistency, called h's neighborhood, and then exclude this neighborhood (including h itself) from the next iteration. We repeat the above process, until no inconsistency can be found, obtaining all the neighborhoods. Note that enlarging a solution is necessary; otherwise, we need over 10^{31} iterations in the worst case.

To get h's neighborhood, we aim to find an ACL rule tuple $\langle sip, dip, sport, dport, proto\rangle$, encoded as m, so that all packets matching the tuple can be handled in the same way as h is, which means:

$$\forall h',\, m(h') \rightarrow h' \in [h]_{FEC} \wedge \bigwedge_{f \in \mathcal{F}_\Omega \cup \mathcal{F}'_\Omega} f(h) = f(h'); \tag{6}$$

We iteratively perform binary search on possible expansions of each field in the tuple (*e.g.*, different IP prefix masks for *sip*), and identify the largest expansion that satisfies Equation (6). The set of packets matched by the resulting tuple forms the neighborhood of h, denoted as $[h]_N$.

Fixing plan generation. After getting all neighborhoods, we seek to generate a fixing plan for each of them. So we need to find out how they should be handled (permitted or denied) on each interface to achieve consistency, and then add high-priority fixing rules on the interfaces.

Specifically, given each neighborhood $[h]_N$, we use SMT solver to get the decision (permit or deny) on $[h]_N$ for each target interface ξ so that the reachability of $[h]_N$ can be consistent with the original. Such a "decision" is defined as a boolean function, $\mathcal{D}_{[h]_N}(\xi)$. Given an interface ξ, $\mathcal{D}_{[h]_N}(\xi)$ returns *TRUE* or *FALSE* to represent whether $[h]_N$ should be permitted or denied by ξ after the update.

According to packet reachability consistency, for each path p, the decision on $[h]_N$ after the update should be the same as before, which is $c_p(h)$. Meanwhile, the updated decision can be represented using $D_{[h]_N}$:

$$c'_p(h) \;,\; \hat{\bigcup_{\xi \in p}} \mathcal{D}_{[h]_N}(\xi); \tag{7}$$

Thus we use the above $c_p(h)$ and $c'_p(h)$ to construct Equation (3), and use an SMT solver to get each interface ξ_j's $\mathcal{D}_{[h]_N}(\xi_j)$, which can tell whether ξ_j should permit $[h]_N$ or not.

After fixing all neighborhoods, we get a final ACL update plan that guarantees packet reachability consistency.

Extensions. During fixing, we can support three additional functions: constraints on ACL placement, optimization for minimal changes, and final ACL simplification.

- *Constraints on ACL placement.* Because the operator can use the **allow** to specify which interfaces can be changed during fixing, such specifications are translated into additional constraints in the SMT solver when solving for Equation (3). In our example, since we are not **allow**ed to fix device C and D which are permitting all traffic, we append the constraints $\bigcirc_{i=1}^{4} \mathcal{D}_{[h]_N}(C_i) = TRUE$ and $\bigcirc_{i=1}^{3} \mathcal{D}_{[h]_N}(D_i) = TRUE$, demanding that the neighborhood be permitted on C and D.
- *Optimization for minimal changes.* When using the SMT solver, we can additionally specify an objective to minimize the number of interfaces we need to fix.
- *Simplifying the final ACL.* The fixing process may render some rules in the original ACL redundant. We design and apply an ACL simplification algorithm that can remove the maximal number of rules from any ACL while preserving the decision model.

A Fixing example. Continuing our example, we know that moving the rule "deny dst 2.0.0.0/8" to interface A_1 causes inconsistency. To fix the inconsistency, we first solve Equation (3) to get a solution that makes this equation true, which is a packet in Traffic 2. Then, the entire Traffic 2 is identified as a neighborhood. Next, our algorithm continues to check whether Equation (3) is satisfiable, finding the other remaining neighborhood, Traffic 1. Finally, we solve the SMT problem for both neighborhoods, starting with Traffic 2. Recall that we want to clean up device C and D, we thus add additional constraints to ensure $D_{[2]_N}$ is always *TRUE* for interfaces on C and D. The SMT solver returns an optimal solution: $D_{[2]_N}(A_2) = FALSE$, and other variables are *TRUE*. Compared with ACLs after the update, the actions on A_1 and A_2 has changed, so that we need to add a rule to permit Traffic 2 in interface A_1, and add a rule to deny Traffic 2 on interface A_2. Similarly, we find that A_1 should add a rule to permit Traffic 1. After fixing, interface A_1's ACL should be "permit dst 1.0.0.0/8, permit dst 2.0.0.0/8, deny dst 1.0.0.0/8, deny dst 2.0.0.0/8, deny dst 6.0.0.0/8, permit all"; A_2's ACL should be "deny dst 2.0.0.0/8." Finally, A_1's ACL is further simplified by removing the first four redundant rules.

5 ACL MIGRATION

We now detail the design of the **generate** primitive, based on an ACL migration task example. Suppose in Figure 1, an operator is asked to remove ACLs from interfaces in $\mathcal{S} = \{A_1, D_2\}$ (called source interfaces) and generate new ACLs at interfaces in $\mathcal{T} = \{C_1, C_2, D_1\}$ (called target interfaces), while preserving packet reachability.

To express this intent, the operator sets the **scope** to all interfaces in Figure 1, **modify** $\mathcal{S}$ to permit all traffic, while **allow**ing new ACLs to be **generate**d at $\mathcal{T}$. Again, because this LAI program does not use **control**, we migrate ACL rules while keeping packet reachability. We describe how to extend **generate** for the desired reachability case in §6.

A strawman solution. To generate ACLs, we need to know whether each packet should be denied or permitted on each interface. A strawman solution is to adopt the fixing approach in §4.2 on all neighborhoods in the entire space. However, this yielded millions of neighborhoods in our network.

Intuition of generate design. We need a more efficient way to divide packets. Since the migration task seeks to replicate the effect of the original ACLs, a natural choice is to use these ACLs themselves to divide packets into a small number of classes (§5.1). By solving the placement problem for each class, we can learn which interface should deny or permit which class (§5.2). In cases where ACL-based classes are too coarse-grained to provide a valid solution, we introduce routing information to loosen placement constraints and make decisions on the more fine-grained classes (§5.3). When all decisions are computed, we can then translate the decisions into ACLs (§5.4).

We now detail the workflow of **generate**.

5.1 Deriving ACL Equivalence Classes

First, we assign all traffic in the given scope Ω to different ACL equivalence classes (AECs), and derive these AECs. An ACL equivalence class, $[h]_{AEC}$—different from the forwarding equivalence class defined in §4.1—means all traffic belonging to the class result in the same ACL decision, exemplified by packet h. In our migration example, the seven types of traffic can be assigned to four AECs, so that we derive these four AECs, as shown in Table 3. Due to the redundancy in ACL usage and the commonality of ACL goals, the growth rate of AECs we experienced is at most polynomial.

Table 3: ACL equivalence class for Figure 1 example.

Class	Traffic	A1	C1	D2
$[1]_{AEC}$	Traffic 1-2	permit	permit	deny
$[3]_{AEC}$	Traffic 3-5	permit	permit	permit
$[6]_{AEC}$	Traffic 6	deny	permit	permit
$[7]_{AEC}$	Traffic 7	permit	deny	permit

5.2 Solving ACL Equivalence Classes

With the derived AECs (*e.g.*, $[h]_{AEC}$) in hand, the goal of this step is to solve the decision function $\mathcal{D}_{[h]_{AEC}}$ for each of the derived AECs to determine how it should be handled by each interface, similar to how $\mathcal{D}_{[h]_N}$ works in §4.2.

Before the migration, $[h]_{AEC}$ is permitted on path p if and only if $c_p(h)$ is TRUE, which we have defined in Equation (1).

After the migration, the source interfaces permit all traffic, while the decision on target nodes is changed based on $\mathcal{D}_{[h]_{AEC}}$. Thus, we define the decision model of interface ξ after the migration as f'_ξ, and the updated path model of path p as c'_p. We have:

$$f'_\xi(h) \triangleq \begin{cases} TRUE & \text{if } \xi \in \mathcal{S} \\ \mathcal{D}_{[h]_{AEC}}(\xi) & \text{else if } \xi \in \mathcal{T} \\ f_\xi(h) & \text{otherwise} \end{cases} \tag{8}$$

$$c'_p(h) \triangleq \hat{\bigcup}_{\xi \in p} f'_\xi(h) \tag{9}$$

Note that this definition for AEC and $f'_\xi(h)$ is based on the migration example where the updated ACLs are permitting all traffic, but it can be easily extended to cover arbitrary updates. The details are omitted due to space constraints.

Solving for AEC-level consistency. To achieve packet reachability for each AEC, $[h_i]_{AEC}$, we use SMT solver to find $\mathcal{D}_{[h_i]_{AEC}}$. Thus h is handled consistently on all paths:

$$\hat{\bigcup}_{p \in \mathcal{P}_\Omega} c_p(h_i) \Leftrightarrow c'_p(h_i) \tag{10}$$

Suppose we wish to compute a solution $\mathcal{D}_6$ for $[6]_{AEC}$ in Table 3. Since class $[6]_{AEC}$ is denied on A_1, it needs to be denied on all paths involving A_1. Thus, $\mathcal{D}_6$ has to be *FALSE* at least on one interface for each path. Thus, by solving the above constraints via SMT solver, we get that class $[6]_{AEC}$ should be denied on all target interfaces, C_1, C_2 and D_1.

5.3 Solving Dataplane Equivalence Class

If we can successfully solve all the derived AECs, we can directly enter the ACL generation phase (§5.4). However, some equivalence classes might be unsolved (*i.e.*, unsat by SMT solver), in which case the constraints need to be loosened by disregarding unused paths. For the migration example, when we try to compute a solution for class $[1]_{AEC}$, which is denied by D_2 (see Table 3), we need to ensure it is denied on path $\langle A_1, A_3, C_1, C_4, D_2, D_3 \rangle$, among which only C_1 is a target interface, so $[1]_{AEC}$ has to be denied by C_1. However, as it is not denied along path $\langle A_1, A_3, C_1, C_3 \rangle$, Equation (10) dictates that it be permitted by C_1 after migration. Thus, there is no valid solution for the ACL equivalence class $[1]_{AEC}$.

Dataplane equivalence class (DEC). Since different packets in the same AEC may be allowed on different paths, they should be solved separately. We, therefore, take into account routing information and split such "unsolved" AECs into dataplane equivalence class (DEC). For a DEC, $[h]_{DEC}$, all packets belonging to this class result in the same decision by ACL and routing configurations, exemplified by packet h. In other words, DEC is working as a conjunction of FEC (defined in §4.1) and AEC (defined in §5.1).

When we take into account the dataplane information, we can derive DECs for Traffic 1 and 2 separately, which we denote as $[1]_{DEC}$ and $[2]_{DEC}$. In this case, $[2]_{DEC}$ can be forwarded from A_2 to B_1, but $[1]_{DEC}$ cannot.

Solving for DEC-level consistency. Solving DECs has two steps. First, for each unsolved AEC, say $[h]_{AEC}$, we divide these AECs into DECs—like the case that we divide $[1]_{AEC}$ into $[1]_{DEC}$ and $[2]_{DEC}$, getting two DECs.

Second, for each derived DEC, we need to solve its corresponding $\mathcal{D}_{[h]_{DEC}}$, via SMT solver, to keep the packet reachability consistency. Specifically, for each $[h_i]_{DEC}$, we iterate all the paths p in Ω and check if p_i allows $[h_i]_{DEC}$ to go through Ω. If so, we put p_i in a set $\mathcal{Y}_{[h_i]_{DEC}}$. We then solve the formula Equation (10) while replacing $\mathcal{P}_\Omega$ with $\mathcal{Y}_{[h_i]_{DEC}}$, thus solving $\mathcal{D}_{[h]_{DEC}}$. If any of DECs does not have a valid $\mathcal{D}_{[h]_{DEC}}$, then there is no valid solution for the given migration intent.

5.4 ACL Synthesis

After getting all the decisions $\mathcal{D}$, we need to convert them to ACL rules on target interfaces, C_1, C_2 and D_1 in our example.

Step 1: Ordering AECs based on the sequence encoding. For each AEC, say $[h_i]_{AEC}$, we use it to "hit" each of target interfaces, say j, in Ω, and check which ACL rule in j can be hit by $[h_i]_{AEC}$. We use the priority of the hit rule as the sequence number. In Table 4a example, $[6]_{AEC}$ traverses the interface A_1, C_1, and D_2, because only these three interfaces have ACLs. As shown in Table 4a, $[6]_{AEC}$ hits A_1's first rule, C_1's second rule, and D_2's third rule, so one entry is generated for $[6]_{AEC}$ by joining the three rules, encoded as 123. Since $[1]_{AEC}$ hits both the first and second rules in D_2, two entries are generated by joining the second rules in A_1 and in C_2, encoded as 221 and 222 respectively. This way, we generate all the sequence encoding numbers based on the AECs (as shown in Table 4b) and sort them accordingly.

Step 2: Computing overlap field. For each row in Table 4b, we compute the intersection of destination IPs of all rules in the sequence encoding. We call this intersection the overlap field. Similarly, we also compute the overlap field of src-ip, src-port, and dst-port. As shown in Table 4, because the first row records A_1's first rule, C_1's second rule, and D_2's third rule, the overlap field of the first row's dst-ip is the intersection of 6.0.0.0/8, all, and all, *i.e.*, 6.0.0.0/8. Thus, we fill 6.0.0.0/8 in the overlap field column for the first row.

Step 3: Synthesizing decisions. We use the $\mathcal{D}_{AEC}$ results obtained from solving ACL equivalence class step to as the decision for each AEC. For example, because $\mathcal{D}_{[6]_{AEC}}(C_1) = FALSE$, $\mathcal{D}_{[7]_{AEC}}(C_1) = FALSE$, $\mathcal{D}_{[1]_{AEC}}(C_1) = TRUE$, and $\mathcal{D}_{[3]_{AEC}}(C_1) = TRUE$, we fill "deny", "deny", "permit", "permit" in the synthesized decision's C_1 column in Table 4b. So far, we can synthesize ACL for C_1 as: "deny dst 6.0.0.0/8, deny dst 7.0.0.0/8, permit dst 1.0.0.0/8, permit dst 2.0.0.0/8, permit all."

Table 4: ACL migration for Figure 1 example: migrating ACLs from A_1 and D_2 to the rest of nodes in this subnet.

	Priority	Action	Dst-IP
A1	1	deny	6.0.0.0/8
	2	permit	all
C1	1	deny	7.0.0.0/8
	2	permit	all
D2	1	deny	1.0.0.0/8
	2	deny	2.0.0.0/8
	3	permit	all

A1 d	[6]AEC
C1 p	Sequence encoding:
D2 p	123
A1 p	[1]AEC
C1 p	Sequence encoding:
D2 d	221
	222
... ...	

(a) Generated sequence encoding. p denotes permit, and d denotes deny.

Step1			Step2	Step3			
Sequence Encoding			AEC	Overlap Field	Synthesized Decision		
A1	C1	D2			C1	C2	D1
1	2	3	[6]	6.0.0.0/8	deny	deny	deny
2	1	3	[7]	7.0.0.0/8	deny	permit	permit
2	2	1	[1]	1.0.0.0/8	permit	permit*	permit
2	2	2	[1]	2.0.0.0/8	permit	permit*	permit
2	2	3	[3]	all	permit	permit	permit

(b) Synthesized ACLs. permit* means the permit is only partial (detailed in §5.4 Step 4)

Step 4: Synthesizing ACL rules for DECs. Finally, we show how ACLs can be synthesized when the decision is made on dataplane equivalence classes. Take C_2 as an example, we learn that it should deny class $[2]_{DEC}$ and class $[6]_{AEC}$. As class $[2]_{DEC}$ belongs to the ACL equivalence class $[1]_{AEC}$, this means part of $[1]_{AEC}$ is denied. Therefore, the ACL synthesized for C_2 in Table 4b is only an intermediate result. We need to insert deny rules before the "permit*" rules to reflect C_2's deny decision on $[2]_{DEC}$. For each "permit*" rule, we identify its intersection with the denied DEC and insert rules before the "permit*" rule to deny the intersections. In our example, the first "permit*" rule does not overlap with $[2]_{DEC}$, thus no action is needed. However, the second rule, "permit* dst 2.0.0.0/8," does overlap with $[2]_{DEC}$, with the intersection being "dst 2.0.0.0/8." Hence, we insert "deny dst 2.0.0.0/8" above the "permit*" rule. In the end, we can generate ACL for C_2 as: "deny dst 6.0.0.0/8, permit dst 7.0.0.0/8, permit dst 1.0.0.0/8, deny dst 2.0.0.0/8, permit dst 2.0.0.0/8, permit all."

5.5 Optimization

While we can correctly generate ACL rules on target interfaces, the current solution may not scale well in some cases. Thus we need to design optimization schemes to greatly reduce run time and the length of generated ACLs. The principle behind our optimizations aims to simplify the complexity of generated ACL rules (*e.g.*, minimize the number of ACL rules and reduce unnecessary ACL rules), thus reducing the number of clauses in SMT encoding.

Generating fewer ACL rules. Our algorithm (§5.4) may generate many redundant ACL rules. For example in Table 4b, it is unnecessary for C_1 to generate three permit rules: "permit dst 1.0.0.0/8," "permit dst 2.0.0.0/8" and "permit all," because "permit all" alone is sufficient. Thus we optimize the number of generated ACL rules.

For any interface, before generating ACL rules, we put all rows in a set R as a candidate rule set. We iteratively check for each rule $r_i \in R$ whether any $r_j \in R$, meets both of the following conditions: 1) r_j's sequence encoding number is lower than r_i's, 2) r_i and r_j have different decisions. If yes, outputting r_i first may render r_j inconsistent. For example, in Table 4b, "deny dst 6.0.0.0/8" (as r_j) is encoded as 123, while "permit all" (as r_i) is 223. Clearly, the two conditions both hold, and "permit all" should not be placed before "deny dst 6.0.0.0/8." Thus, we remove "permit all" from R. After all such r_i are removed from R, we pick one rule (say r_k) from R that can cover the most rules in R, and generate it. For the rules not covered, we repeat the above process until all rules are either generated or covered.

Grouping ACL rules before sequence encoding. We optimize step 1 in ACL generation (§5.4), by grouping ACL rules at source interfaces to reduce the number of generated rows in Table 4b. The insight is that consecutive rules with the same decision can be regarded as one, since they belong to the same ACL equivalence group and are interchangeable in position. For example, in Figure 4a, on interface D_2, we can group "deny dst 1.0.0.0/8" and "deny dst 2.0.0.0/8" into one item. When combining groups, we also take advantage of the fact that adjacent non-overlapping rules can switch position regardless of decision type, thus enabling more aggressive grouping. *This optimization can yield 98.6% drop in number of generated items per interface, significantly improving efficiency.*

ACL search tree. Once we put ACLs into groups, our overlap field computation in step 2 (§5.4), becomes computing the overlap between two groups rather than computing the overlap between two rules. We can speed up this process using a search tree. When checking for overlapping fields across two trees, we work only on subtrees that overlap with each other, thus significantly reducing cost. We omit the details due to space constraints.

6 UPDATING REACHABILITY

Given **control** requirements, our system needs to achieve desired reachability consistency, which is a modification on the original reachability. The primitive **control** is used to specify the reachability updates needed between a pair of interfaces. For example, for a source and destination pair $\langle A_1, C_3 \rangle$, the intent "**control** $A_1 \rightarrow C_3$ **open** dst 6.0.0.0/8" expresses that the packets going to 6.0.0.0/8 should be permitted to go from A_1 to C_3. Similarly, if the intent is to **isolate**, the specified traffic should not be able to go through the given interface pair. When multiple intents for the same pair of interfaces overlap with each other, their priority is determined by the specification order. Thus **maintain** can take precedence over other intents, and protect the original reachability from the current update. For example, an operator may specify the intent as "**control** $A_1 \rightarrow C_3$ **maintain** dst 7.0.0.0/8" followed by "**control** $A_1 \rightarrow C_3$ **isolate** dst all". This would mean for all paths between A_1 and C_3, if any packet to 7.0.0.0/8 was originally permitted, it should still be permitted after the update, while the traffic to all other destination IPs should be blocked.

The **control** primitive can be supported by simply extending existing designs of **check**, **fix** and **generate**. In principle, because **control** does not affect the construction of c'_p, we still construct c'_p like how we did for the packet reachability consistency. For c_p, since c_p is responsible for representing the desired reachability, we

should adapt c_p based on the intent specified by **control**. With the updated c_p in hand, we can follow the methodologies used in §4 and §5 to achieve the desired reachability consistency.

We now detail how to obtain the updated c_p. We start by parsing all primitives (**isolate**, **open**, and **maintain**), and encode the update decisions into a decision model $r(h)$ with three different outputs "isolate," "open" and "maintain," respectively. Each path has its own function r_p based on its starting and ending interfaces. If no change is specified, then decisions are maintained for all traffic. Thus, we create an updated path model using r_p as a constraint on the original c_p, and transform the packet reachability consistency constraints to that of *desired reachability consistency*. In addition, a few details need to be adapted in the original design. We need to take all r functions into consideration when getting neighborhoods for **fix** in §4.2, and when deriving AECs for **generate** in §5.1. For **check**, "isolate" and "open" related prefixes need to be taken into account when computing deferential rules in §4.1.

7 EXPERIENCE WITH JINJING

Jinjing has been deployed in Alibaba's global WAN for several months. It has successfully guided our operators in avoiding a lot of potential critical service disruptions while managing network-wide ACL configurations. This section first describes deployment challenges, and then shows three real scenarios, from a small-scale but frequently-happened scenario to a large-scale scenario Jinjing was used in.

Deployment challenges. Jinjing's deployment was faced with many challenges, such as incomplete topology data, inaccurate dependency collection, and tricky data formats. Given space constraints, we only discuss the main challenge: data sources Jinjing relies on, (*e.g.*, routing information and parsed configuration format), are incomplete or inaccurate in practice. For example, routers in our WAN are provided by different vendors. They not only have different configuration formats, but also contain various implicit routing behaviors. This inaccurate information would significantly affect the capability of Jinjing. To address this challenge, we develop an internal auditing tool to timely monitor and manually repair the quality of the data Jinjing relies on.

Scenario 1: Isolating service area. Managing (*e.g.*, isolating or opening) service traffic is one of the most common uses of Jinjing. In one of our real-world traffic isolation events, a new service S was deployed and some IP prefix (say, 1.2.0.0/16) is assigned to S. Our operators want to isolate the traffic between S and device R_3, which is a gateway managing an important private subnet.

Between S and R_3, there are two routers R_1 and R_2. Traffic between S and R_3 are allowed to access both R_1 and R_2. The operator cannot directly add a deny rule on R_3 because it may have side effects on un-recycled IP segments in R_3's internal network. Thus, our operators write the following intent using LAI.

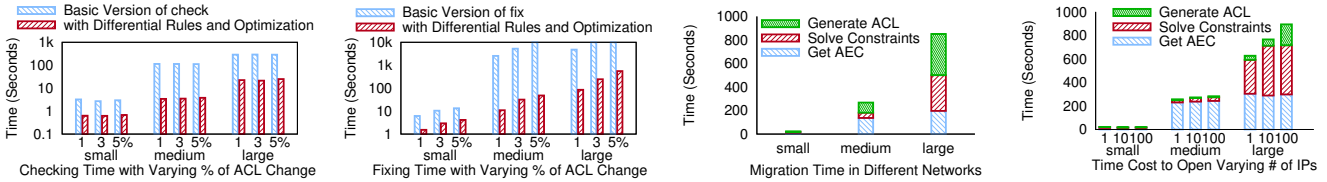
```
1. scope R1:*, R2:*, R3:*
2. allow R1:*-in, R2:*-in, R3:*-in
3. control R1:*,R2:* -> R3:*-out isolate from 1.2.0.0/16
4. control R3:*-in -> R1:*,R2:* isolate to 1.2.0.0/16
5. generate
```

As shown above, **scope** and **allow** specify that we should generate ACL rules on the ingress interfaces of R_1, R_2 and R_3 without touching any other interface or device. Line 3 isolates the traffic from the subnet (managed by R_3) to S, and line 4 applies to the traffic in the reverse direction. Jinjing generates a valid plan within a minute. The plan adds several lines of ACL rules to the ingress interface of R_3. We also asked our operators to manually write ACL rules for this objective. It took them 100× more time to produce a plan, because they need to check for potential side effects on all paths. Because the above ACL update tasks occur hundreds of times each month, Jinjing significantly reduces our management time.

Scenario 2: Hidden complexities in moving ACLs from ingress to egress. Our global WAN employs many cells communicating in a complex way, each using tens of gateways to control the traffic that enters/leaves the cell. In an important network upgrade event, our operators needed to relocate the ACLs of all gateways in the hundreds of the cells from their ingress interfaces to their egress interfaces. Such a seemingly innocuous move may block the traffic between routers within this cell since the communication within the cell only go through the gateways' egress interfaces. Essentially, the operators need to identify what traffic could be affected by the migration, and decide how to offset the effect without introducing new problems. Given the complexity and scale of our network—hundreds of cells containing thousands of routers, it is difficult for our operators to correctly complete this task. Initially, our operators spent multiple weeks manually producing a migration plan. Before committing the plan, they used Jinjing's **check** command to check whether the update plan retains the packet reachability. Inconsistency was reported within a minute by Jinjing, suggesting that some traffic within many cells would be blocked after moving. Then, within a few minutes, fixing plan were generated by Jinjing. Without Jinjing, an incorrect migration like this would have caused critical service disruption. In this scenario, it is unnatural for the operators to directly generate an ACL update plan via **generate**, because the generated ACLs may look different from the original ACL rules, and make those ACL configurations hard to be maintained.

Scenario 3: Migrating ACLs from a large scope of routers. A third experience we would like to share is an event where our operators needed to migrate the ACL rules out of a layer of core routers in order to reassign them as provider-edge (PE) routers supporting the MPLS protocol. This migration involves 30% of all the routers in our WAN, each of which managing thousands of routing paths and ACL rules. A network upgrade like this can reduce expenses by replacing those core routers with much cheaper ones with no ACL support. This task is of greater challenge than the former two scenarios discussed above, since 1) the paths to a gateway are controlled by multiple ACLs, and 2) different paths may overlap with each other. Jinjing returned a secure migration plan in a minute. Before Jinjing was developed, our operators typically spent half a month in planning how to securely migrate the ACLs without disrupting service.

Other experience: ACL rules generated by Jinjing vs. by the operator. We comment briefly on how Jinjing-generated ACL rules differ from ACL rules manually written by our operators. In most cases, they are quite similar. For example, both our operators and Jinjing prefer to add "permit" to allow certain traffic,



(a) Consistency checking (b) Fixing inconsistency (c) Generate migration plan (d) Reachability control (open)

Figure 4: Time cost of Jinjing operations in different network settings.

rather than removing "deny" from the original ACLs. However, they differ in some other cases. Jinjing employs a set of optimization approaches to compress generated ACL rules; while the operators are inclined to write redundant rules for easier comprehension.

8 NETWORK-WIDE EXPERIMENTS

To understand the performance of Jinjing's primitives, we carefully choose representative sub-networks in our WAN as testbeds to evaluate the scalability of the primitives, **check**, **fix** and **generate**, for both **modify** and **control**.

Experiment setup. We take three sub-networks of different sizes from our network (8%, 30%, and 80% of our WAN), representing small, medium and large networks respectively. We have no access to the remaining 20% due to company restrictions. They all have layered topology and are connected to an external backbone network. The ACLs and IP prefixes used in the experiments are taken from real configurations in the selected networks, with thousands of ACL rules placed across multiple layers. All our experiments are run on a server with 128GB RAM and a 2.9GHz 4-core processor.

Checking ACL update plans and fixing. To evaluate the performance of **check** and **fix**, we generate ACL update plans by randomly perturbing 1%, 3%, and 5% of the rules in each router, and use **check** to check whether the packet reachability is preserved in the updated plans. In the case that the packet consistency is violated, we run the **fix** primitive.

As shown in Figure 4a, the turnaround time is not affected by the update percentage. This is reasonable, because once a single inconsistency is detected, **check** warns and returns this violation. We also observe that the turnaround time is one order of magnitude more efficient in computation than the basic version without differential rules The **check** with differential rules takes less than one minute for the largest setting.

Figure 4b presents the turnaround time of the **fix** operation for different sub-networks. We observe increases in turnaround time with higher ratio of perturbation, due to the increase in fixes required. The use of differential rules and optimization speeds up fixing by two orders of magnitude in both medium and large subnets. Taking at most 10 minutes in total, **check** and **fix** work well in assisting daily updates.

ACL migration. To evaluate the scalability of **generate** primitive, we use the common scenario of ACL migration, where we move all ACLs from middle layer to lower layers and examine the time cost for each migration operation, as shown in Figure 4c. Three steps are involved when performing migration: 1) deriving all ACL equivalence classes 2) solve for each class and identify DECs when necessary, and 3) generate ACLs from decisions on AECs and DECs.

As subnets increase in scale, migrations become more and more costly as the generated ACL has to represent the decision of different ACLs on different paths. Our optimization is able to mitigate such complexity to some degree, reducing the run time and the length of the generated ACL by two orders of magnitude, with the entire operation taking less than 15 minutes for the largest network.

Reachability control. We also examine the scalability of our algorithm when dealing with the **control** primitive. Due to space constraint, we only show the result for **control open** with the command **generate**. In our experiment, we apply the **control open** intent to a randomly selected set from 1000+ IP prefixes announced by each lower level device. To see how the LAI program size affects performance, we choose three different settings, selecting 1, 10 or 100 IP prefixes respectively in each device. The time cost for each scenario is detailed in Figure 4d. Compared to migration, the time cost in deriving AEC is slightly higher since the reachability control models, r, are taken into account when generating AEC, as discussed in §6. On the other hand, the time cost in generating ACL is significantly lower since this scenario benefits more from our optimization algorithm, which is able to reduce the length of the generated ACLs by up to four orders of magnitude.

Expressiveness of LAI. Table 5 shows the number of lines to write the LAI program for our above experiments. Even with the largest network setting, **check**, **fix** and **generate** for migration only need about ten lines. Thus, using LAI is simple.

9 DISCUSSION

We discuss the details of important practical issues in this section.

Why our optimizations enable Jinjing to scale well? Generally, any SMT solver shares the worst case complexity of at least $O(2^n)$, where n is the number of boolean variables. One may confuse about how to measure the complexity of a verification problem, because all algorithms unavoidably involve n variables, *e.g.*, 104 variables for 5-tuples. However, in practice, the complexity of an SMT problem is described as the number of recursive calls when running a DPLL [8, 9] based SMT solver [30]. Less recursive calls can make a problem solved faster.

Table 5: LAI program line count in experiments.

Network	check & fix	migration	open 1	open 10	open 100
Small	O(1)	O(1)	O(1)	O(10)	O(50)
Medium	O(1)	O(1)	O(10)	O(50)	O(100)
Large	O(10)	O(10)	O(10)	O(100)	O(500)

In fact, directly or indirectly, all optimizations in Jinjing aim at reducing the recursive calls. For example, differential rules reduce the chance of backtracking by reducing the clauses, which significantly reduces recursive calls. Similarly, our new decision model trades a larger width of DPLL searching tree for a smaller searching depth. We observed recursive calls reduced by about $O\ \frac{k}{\log k}$ times in experiments, where k is the length of the largest ACL. This is the reason why Jinjing scales well.

Can Jinjing work if the traffic class is unknown? AECs are always known, because we can directly derive them from ACLs. So we only discuss the cases when we cannot get FECs or DECs. For **check** primitive, we cannot specify the traffic we verify without FECs. Instead, we can directly verify all traffic, *i.e.*, `0.0.0.0/0`, on each ACL individually, which is a sufficient condition (but much stronger) for the reachability consistency. Note that it may cause false positive, and possibly bring too much computation to the **fix** primitive. As described in §5.3, the **generate** primitive can work with only AECs. Some equivalence classes might be unsolved, because stronger conditions can make an SMT problem unsatisfiable.

Why do prior work on verification or synthesis work on firewalls not work in our scenario? While firewalls provide standalone access control, in-network ACLs filter traffics in a distributed manner, so that we need to handle not only what to do, but also where to do. An in-network ACL configuration update requires a global view of the network, including topologies, routing tables and traffic. Otherwise, a seemingly innocent rule can easily harm other traffic unexpectedly. On the contrary, firewall verification and repair efforts mainly focused on end-host control rather than in-network information like routing and topology information.

10 RELATED WORK

ACL placement and migration. Sung *et al.* [32] proposed a heuristic algorithm for placing ACL rules on routers based on operators' intent. Compared with Jinjing, we target different problems. First, their approach does not support operations like reachability checking and ACL migration, which are highly needed by our ACL management. Second, their approach is only scalable to a small local area network. Finally, the correctness of their approach is not provably guaranteed by a provable checker (like SMT solver).

Hajjat *et al.* [16] focused on automatically migrating existing enterprise applications' ACL rules to a third-party cloud platform, while preserving the original security property. Zhang *et al.* [38] proposed a linear-programming algorithm to optimize ACL placement based on various constraints. Yoon *et al.* [35] designed a heuristic algorithm to generate routing structures that minimize the rules needed by each gateway. Nelson *et al.* [26] proposed an approach to migrate the configuration from enterprise network to an SDN network setting. Compared with Jinjing, the above efforts target different problems from Jinjing's.

Firewall management. State-of-the-art firewall management work mainly falls into two categories. First, many firewall verification tools [5, 19, 23, 36] were developed to detect the faulty rules that violate the operators' intent; however, they only focused on checking ACL rules on single firewall, rather than a distributed setting. Second, firewall repairing tools [7, 17] fix faulty policies in a single node. It's hard to extend them to distributed setting, since those tools fail to take into account the routing information.

Network configuration synthesis. SyNET [10] and ConfigAssure [25] focused on offering the operators general, network-wide configuration synthesis systems, but they are hard to scale in cloud-scale networks.

Propane [3] and PropaneAT [4] can synthesize BGP-specific configurations from scratch; on the contrary, NetComplete [11] can automatically complete "half-baked" BGP and OSPF configurations by filling the configuration "holes" based on the operators' intent. These protocol-specific configuration synthesis techniques cannot be straightforwardly extended to synthesize ACL-level configurations, because their synthesis algorithms heavily rely on protocol-specific features, such as BGP update propagation.

Zhang *et al.* [39] proposed a technique capable of detecting redundancies of a single-node firewall's policies, checking the equivalence between two given firewalls' policies, and synthesizing the optimal policies for a given firewall (*e.g.*, minimizing the number of firewall policies while keeping the policy correctness). This approach targets totally different problem from Jinjing.

Network configuration verification. A group of works proposed to verify the correctness of network configuration—*i.e.*, whether a given network configuration meets the operator's intent. They include: control plane verification [1, 12, 13, 15, 29, 34], dataplane verification [3, 18, 21, 22, 24, 27, 31], complex network verification [28, 33], and dataplane implementation [6, 37]. These systems are designed for verification only, so that they cannot perform fixing or synthesis. SecGuru [20] is the closet to Jinjing. While SecGuru aims to verify the policy correctness of ACLs and firewalls, it is only focused on single-node model rather than a distributed setting, because its verification does not take network topology and routing paths into account. In addition, SecGuru cannot fix or synthesize ACL or firewall policies.

11 CONCLUSION

We have introduced Jinjing for automatically and safely updating ACL configurations in Alibaba's global WAN. Jinjing offers an intent language for the operators to express their ACL update objectives, and Jinjing can generate the update plans that satisfy their intent. The core of Jinjing is a set of novel verification and synthesis algorithms scalable to the large network. We have presented the real-life experience that our network operators used Jinjing to prevent mismanagement issues, and experiment results that demonstrate Jinjing is scalable.

Ethical concerns. This work does not raise any ethical issues.

Acknowledgments

We thank our shepherd, Aurojit Panda, and the anonymous reviewers for their insightful comments. We also thank Ang Chen and Pengyu Zhang for their valuable feedback on earlier drafts of this paper. Bingchuan Tian and Chen Tian are supported in part by the National Key R&D Program of China 2018YFB1003505 and the National Natural Science Foundation of China under Grant Numbers 61772265. Haitao Zheng and Ben Y. Zhao are supported in part by NSF grants CNS-1527939 and CNS-1705042.

REFERENCES

[1] Beckett, R., Gupta, A., Mahajan, R., and Walker, D. A general approach to network configuration verification. In *ACM SIGCOMM (SIGCOMM)* (2017).

[2] Beckett, R., Gupta, A., Mahajan, R., and Walker, D. Control plane compression. In *ACM SIGCOMM (SIGCOMM)* (2018).

[3] Beckett, R., Mahajan, R., Milstein, T. D., Padhye, J., and Walker, D. Don't mind the gap: Bridging network-wide objectives and device-level configurations. In *ACM SIGCOMM (SIGCOMM)* (2016).

[4] Beckett, R., Mahajan, R., Milstein, T. D., Padhye, J., and Walker, D. Network configuration synthesis with abstract topologies. In *38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (2017).

[5] Brucker, A. D., Brügger, L., Kearney, P., and Wolff, B. Verified firewall policy transformations for test case generation. In *International Conference on Software Testing, Verification and Validation (ICST)* (2010).

[6] Cardwell, N., Cheng, Y., Brakmo, L., Mathis, M., Raghavan, B., Dukkipati, N., Chu, H.-k. J., Terzis, A., and Herbert, T. Packetdrill: scriptable network stack testing, from sockets to packets. In *USENIX Annual Technical Conference (ATC)* (2013).

[7] Chen, F., Liu, A. X., Hwang, J., and Xie, T. First step towards automatic correction of firewall policy faults. *ACM Transactions on Autonomous and Adaptive Systems (TAAS) 7* (2012).

[8] Davis, M., Logemann, G., and Loveland, D. A machine program for theorem-proving. *Communications of the ACM 5*, 7 (1962), 394–397.

[9] Davis, M., and Putnam, H. A computing procedure for quantification theory. *Journal of the ACM (JACM) 7*, 3 (1960), 201–215.

[10] El-Hassany, A., Tsankov, P., Vanbever, L., and Vechev, M. T. Network-wide configuration synthesis. In *29th International Conference on Computer Aided Verification (CAV)* (2017).

[11] El-Hassany, A., Tsankov, P., Vanbever, L., and Vechev, M. T. NetComplete: Practical network-wide configuration synthesis with autocompletion. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2018).

[12] Fayaz, S. K., Sharma, T., Fogel, A., Mahajan, R., Millstein, T., Sekar, V., and Varghese, G. Efficient network reachability analysis using a succinct control plane representation. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2016).

[13] Fogel, A., Fung, S., Pedrosa, L., Walraed-Sullivan, M., Govindan, R., Mahajan, R., and Millstein, T. A general approach to network configuration analysis. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2015).

[14] Gember-Jacobson, A., Akella, A., Mahajan, R., and Liu, H. H. Automatically repairing network control planes using an abstract representation. In *26th Symposium on Operating Systems Principles (SOSP)* (2017), pp. 359–373.

[15] Gember-Jacobson, A., Viswanathan, R., Akella, A., and Mahajan, R. Fast control plane analysis using an abstract representation. In *ACM SIGCOMM (SIGCOMM)* (2016).

[16] Hajjat, M. Y., Sun, X., Sung, Y. E., Maltz, D. A., Rao, S. G., Sripanidkulchai, K., and Tawarmalani, M. Cloudward bound: Planning for beneficial migration of enterprise applications to the cloud. In *ACM SIGCOMM (SIGCOMM)* (2010).

[17] Hallahan, W. T., Zhai, E., and Piskac, R. Automated repair by example for firewalls. In *Formal Methods in Computer Aided Design (FMCAD)* (2017).

[18] Horn, A., Kheradmand, A., and Prasad, M. R. Delta-net: Real-time network verification using atoms. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (Mar. 2017).

[19] Hwang, J., Xie, T., Chen, F., and Liu, A. X. Fault localization for firewall policies. In *International Symposium on Reliable Distributed Systems (SRDS)* (2009).

[20] Jayaraman, K., Bjørner, N., Outhred, G., and Kaufman, C. Automated analysis and debugging of network connectivity policies. In *Technical Report MSR-TR-2014-102* (2014).

[21] Kazemian, P., Varghese, G., and McKeown, N. Header space analysis: Static checking for networks. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2012).

[22] Khurshid, A., Zhou, X., Zhou, W., Caesar, M., and Godfrey, P. B. VeriFlow: Verifying network-wide invariants in real time. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2013).

[23] Liu, A. X. Formal verification of firewall policies. In *International Conference on Communications (ICC)* (2008).

[24] Lopes, N. P., Bjørner, N., Godefroid, P., Jayaraman, K., and Varghese, G. Checking beliefs in dynamic networks. In *12th USENIX Symposium on Networked System Design and Implementation (NSDI)* (2015).

[25] Narain, S., Levin, G., Malik, S., and Kaul, V. Declarative infrastructure configuration synthesis and debugging. *J. Network Syst. Manage. 16*, 3 (2008), 235–258.

[26] Nelson, T., Ferguson, A. D., Yu, D., Fonseca, R., and Krishnamurthi, S. Exodus: Toward automatic migration of enterprise network configurations to SDNs. In *1st ACM SIGCOMM Symposium on Software Defined Networking Research (SOSR)* (2015).

[27] Panda, A., Argyraki, K., Sagiv, M., Schapira, M., and Shenker, S. New directions for network verification. In *LIPIcs-Leibniz International Proceedings in Informatics* (2015), vol. 32, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[28] Panda, A., Lahav, O., Argyraki, K. J., Sagiv, M., and Shenker, S. Verifying reachability in networks with mutable datapaths. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2017).

[29] Quoitin, B., and Uhlig, S. Modeling the routing of an autonomous system with C-BGP. *IEEE Network 19*, 6 (2005), 12–19.

[30] Selman, B., Mitchell, D. G., and Levesque, H. J. Generating hard satisfiability problems. *Artificial intelligence 81*, 1-2 (1996), 17–29.

[31] Stoenescu, R., Popovici, M., Negreanu, L., and Raiciu, C. Symnet: Scalable symbolic execution for modern networks. In *ACM SIGCOMM (SIGCOMM)* (Aug. 2016).

[32] Sung, Y. E., Rao, S. G., Xie, G. G., and Maltz, D. A. Towards systematic design of enterprise networks. In *ACM CoNEXT (CoNEXT)* (2008).

[33] Velner, Y., Alpernas, K., Panda, A., Rabinovich, A., Sagiv, M., Shenker, S., and Shoham, S. Some complexity results for stateful network verification. In *22nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)* (2016).

[34] Wang, A., Jia, L., Zhou, W., Ren, Y., Loo, B. T., Rexford, J., Nigam, V., Scedrov, A., and Talcott, C. L. FSR: formal analysis and implementation toolkit for safe interdomain routing. *IEEE/ACM Transactions on Network (ToN) 20*, 6 (2012), 1814–1827.

[35] Yoon, M., Chen, S., and Zhang, Z. Minimizing the maximum firewall rule set in a network with multiple firewalls. *IEEE Transactions on Computers 59* (2010).

[36] Yuan, L., Mai, J., Su, Z., Chen, H., Chuah, C., and Mohapatra, P. Fireman: A toolkit for Firewall modeling and analysis. In *IEEE Symposium on Security and Privacy (IEEE S&P)* (2006).

[37] Zaostrovnykh, A., Pirelli, S., Pedrosa, L., Argyraki, K., and Candea, G. A formally verified NAT. In *ACM SIGCOMM (SIGCOMM)* (2017).

[38] Zhang, S., Ivancic, F., Lumezanu, C., Yuan, Y., Gupta, A., and Malik, S. An adaptable rule placement for software-defined networks. In *44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)* (2014).

[39] Zhang, S., Mahmoud, A., Malik, S., and Narain, S. Verification and synthesis of firewalls using SAT and QBF. In *20th IEEE International Conference on Network Protocols (ICNP)* (2012).