Validating Datacenters At Scale

Karthick Jayaraman, Nikolaj Bjørner, Jitu Padhye, Amar Agrawal, Ashish Bhargava, Paul-Andre C Bissonnette, Shane Foster, Andrew Helwer, Mark Kasten, Ivan Lee, Anup Namdhari, Haseeb Niaz, Aniruddha Parkhi, Hanukumar Pinnamraju, Adrian Power, Neha Milind Raje, Parag Sharma
Microsoft
networkverification@microsoft.com

ABSTRACT

We describe our experiences using formal methods and automated theorem proving for network operation at scale. The experiences are based on developing and applying the SecGuru and RCDC (Reality Checker for Data Centers) tools in Azure. SecGuru has been used since 2013 and thus, is arguably a pioneering industrial deployment of network verification. SecGuru is used for validating ACLs and more recently RCDC checks forwarding tables at Azure scale. A central technical angle is that we use local contracts and local checks, that can be performed at scale in parallel, and without maintaining global snapshots, to validate global properties of datacenter networks. Specifications leverage declarative encodings of configurations and automated theorem proving for validation. We describe how intent is automatically derived from network architectures and verification is incorporated as prechecks for making changes, live monitoring, and for evolving legacy policies. We document how network verification, grounded in architectural constraints, can be integral to operating a reliable cloud at scale.

CCS CONCEPTS

• **Networks** → **Network management**; **Network monitoring**; *Cloud computing*; • **Computing methodologies** → **Model verification and validation**; • **Computer systems organization** → **Reliability**; **Availability**;

KEYWORDS

Availability, cloud computing, formal verification, network management, network monitoring, network verification, reliability

ACM Reference Format:
Karthick Jayaraman, Nikolaj Bjørner, Jitu Padhye, Amar Agrawal, Ashish Bhargava, Paul-Andre C Bissonnette, Shane Foster, Andrew Helwer, Mark Kasten, Ivan Lee, Anup Namdhari, Haseeb Niaz, Aniruddha Parkhi, Hanukumar Pinnamraju, Adrian Power, Neha Milind Raje, Parag Sharma. 2019. Validating Datacenters At Scale. 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, 14 pages. https://doi.org/10.1145/3341302.3342094

SIGCOMM '19, August 19–23, 2019, Beijing, China

ACM ISBN 978-1-4503-5956-6/19/08…$15.00
https://doi.org/10.1145/3341302.3342094

1 INTRODUCTION

Routing and network access restrictions in today's hyperscale cloud datacenters involves hundreds of thousands of routers and millions of end-points. Conceptually, correctness of configurations that control routing and network access is a global property: *Are all available shortest paths available for routing? Is access from one group of devices to another ensured/prevented?* Specifying and enforcing such properties appears on the surface as an insurmountable challenge: how does one extract specifications that capture the intended operation, coined as *intent* [25], of these networks, and how can extracted intent be checked efficiently? Several approaches inspired by the paradigm of treating *networks as programs* [44] have emerged in recent years to address reachability and access restrictions. These include tackling scalability through network-optimized data-structures [22, 23, 47, 50], methods for handling incremental updates [21, 23, 50], and discovering structural properties of datacenters, such as symmetries [4, 41]. The question of how intent is captured is also an area with several approaches, noteworthy using template properties, a.k.a. network *beliefs* [30], as labels that can be applied to networks.

Our experience with deploying network verification in Azure suggests that there is another useful perspective for verifying reachability. We claim that network intent for a structured datacenter network, can be derived from its architecture. Thus, intent is available in databases that maintain a description of network topology and address locality. Verification methods, in turn, can be *localized* to one device at a time, in isolation, enabling scalability. ACL checking and checking forwarding tables, discussed in depth in this paper, are prime scenarios of network verification using local verification techniques. RCDC can check all-pairs of redundant routes in a datacenter with up to 10^4 routers in less than 3 minutes on a single CPU. In comparison, sophisticated and highly tuned methods for incremental checking forwarding behavior, including [21], reports on experiments with networks having between 12 and 316 nodes with overhead in milliseconds for incremental checks, yet from cold start require $O(|V|^3)$ solver calls for the all pair shortest path reachability problems. This addresseses useful properties, but doesn't directly address equal cost multi-path routing properties. Libra [50] reports scalability to 10^4 switches, and relies on MapReduce jobs among 50 machines to bring global verification of properties down to a minute. To reduce the number of nodes, $|V|$, to a feasible range, symmetry reduction techniques may apply, but an available reduction [41] taking 100 minutes for a datacenter with fewer than 400 routers renders this avenue several order of magnitudes slower than local checks. The resources required for local checks are trivial in comparison to global approaches.

We view locality analogously to program verification based on programs annotated with inductive loop invariants that can be checked locally by verifying proof obligations known as Hoare triples. In contrast, symbolic model checking methodologies address the case where programs are not already annotated with inductive loop invariants and the tools have to work much harder to synthesize the invariants. With the locally discharged verification conditions, correctness of the entire program is derived in a compositional way. Similarly, global properties of the datacenter networks are derived from how the routers are connected. We claim that locality derives from the necessity that datacenter designs adhere to the law of parsimony that allows devices to be configured in modular and uniform ways. Thus, we are not faced with an arbitrary verification problem, but a scenario where the contracts on routers follow a fixed set of rules. These rules derive from the datacenter architecture which remains essentially fixed. The state of the network may fluctuate, for example as a result of link failures, software, firmware or hardware bugs, or manual configuration changes. The contracts derived from the architecture, though, are expected to hold across such state changes. For example, we expect all devices to have at least a minimum set of uplinks available. This is a fundamental observation that allows to significantly reduce the set of moving parts that have to be addressed, and ultimately enables our local methodology. Checking ACLs and forwarding share several traits: policies are checked using symbolic techniques, either simple SMT queries or lookups into optimized hash-tries, they derive from the dataplane state, and they are deployed as services in a change validation workflow.

We also argue, as documented by our experiences, that network verification is a necessary part of deploying modern hyperscale datacenters. Verification would also be useless unless there is a process for remediation. To this end, we describe our experiences with routing and network access restrictions over nearly six years of deployments and over different incarnations of Azure's designs.

Contributions. The paper makes the following contributions:

(1) **Automatic intent extraction**: We describe a methodology for automatically deriving intent for structured datacenter networks from architectural metadata comprising topology and address locality.
(2) **Local validation**: A novel technique for decomposing end-end reachability invariants to a set of local contracts.
(3) **Design and implementation** of an industrial deployment for continuously verifying routing and network access restrictions of a hyperscale network.
(4) **Experiences**: We describe experiences from a set of use cases for managing forwarding policies and network access control. We have deployed our techniques for live monitoring of production state and as prechecks for ensuring that the impact of changes is along the intent.

The contributions are described in a context of four themes:

What defines reality and how is it captured? In our setting, reality is given as configurations that reside on network devices. The forwarding information base (FIB) determines packet forwarding behavior. Access Control Lists (ACLs) control network access restrictions.

How do we learn intent? We automatically extract intent based on facts about our network topology and architecture. Azure has a metadata service that maintains facts such as the IP prefixes hosted in the top-of-rack switch routers, the details of the neighbors, and how the BGP sessions are configured between routers.

How is reality validated against intent? Our approach is based on *locally validating* device state in the backdrop of global network properties that follow from the network design. Notably, validation does not assume or require forming global snapshots. We claim that correctness of a structured datacenter is based on local properties of routers. This ensures scalability as checks can be parallelized.

Are results actionable? Verification produces reports that are used to trigger alerts or automatic remediation actions. Reports that require human inspection are processed by raising alerts with appropriate severity for network operators. Error reports that are from well-understood failure modes trigger automatic remediation.

The experiences provided in this paper aim to shed light on the role of verification in a DevOps cloud scenario. Our efforts document how this setting has its own distinguishing characteristics compared to, for instance, the role of verification in hardware designs and static analysis of software. Our setting involves verification of live systems that change through upgrades and new datacenter designs and is managed by operators who in part have built components. We document how verification is integrated to gate deployments. Yet, they share one commonality: our approach for extracting forwarding intent is based on specifications baked into a systems architecture, similar to how an instruction set specification can be said to capture the intent of how a CPU should operate. Note that network verification also broadly covers areas that are orthogonal to ours, such as program verification and symbolic simulation of P4 programs [28, 43], NATs [49], middle-boxes [39], and control plane verification [3, 13], and finally verification of WAN properties where our assumptions of uniform designs may not hold.

Our work or experiences do not raise any ethical issues.

2 FORWARDING BEHAVIOR

Azure has a hyperscale network comprising hundreds of thousands of network devices. These translate to billions of reachability invariants to verify that all datacenter traffic preserves our intent. We describe our approach for formulating intent, reality, and our technique for efficiently validating reality against intent. We deployed these techniques in RCDC, a system for continuously monitoring drift of forwarding policies. We also developed pre checks leveraging a high-fidelity network emulator for verifying that all changes to the network leave the devices in a state that preserves our intent. We describe the experiences of operating both these systems.

2.1 Datacenter Network Architecture

We first provide a brief primer on the Azure datacenter network architecture. It builds on previous work [17, 24] that describes the architecture and design rationale.

The Azure datacenter network uses a hierarchical Clos topology leveraging commodity switch hardware. It uses External BGP (EBGP) as the routing protocol, and equal-cost multipath (ECMP) for load balancing traffic across all links. These aspects combined

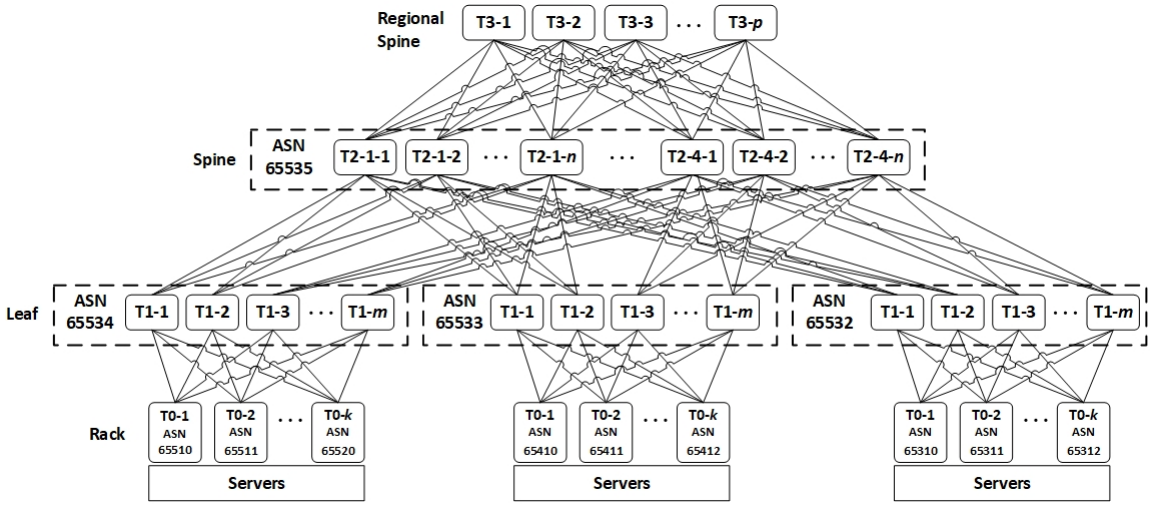

Figure 1: Datacenter Topology Diagram

offer high bandwidth and low latency for both east-west and north-south traffic.

Figure 1 illustrates Azure's datacenter network topology. The *top-of-rack* (ToR) switches connect a set of servers hosted in a rack. A number of top-of-rack switches are connected together by a set of aggregating switches referred to as the *leaf* switches. We will refer to the set of racks that are connected together as a *cluster*. Leaf switches form the cluster boundary. Leaf switches are in turn connected together by a set of aggregating switches referred to as the *spine* switches. Spine switches form the boundary for the datacenter. Spine switches connect the datacenter network to the Azure regional spine network.

All network devices are configured to establish EBGP sessions over direct point-to-point links interconnecting them. The other aspect of the EBGP configuration is the ASN allocation scheme. Azure's ASN allocation scheme uses private ASN numbers to avoid conflicts with the Internet. The scheme assigns a unique ASN number to all spine devices serving a datacenter, all leaf devices serving a cluster, and each ToR switch. We reuse ToR ASN numbers such that they are unique within a cluster. Therefore, upstream BGP sessions on the ToR are configured to accept BGP announcements for prefixes hosted in other ToRs with the same ASN number.

The top-of-rack switches are configured to announce their VLAN prefixes to their neighboring leaf devices. The leaf and spine devices are configured to announce the received prefixes to their neighbors; they do not use route aggregation because such aggregations can result in black-holing of traffic due to a single-link failure. The regional spine devices are configured to strip off the private ASNs from the AS_PATH property when relaying the routes received from the spine devices. This is done to prohibit ASN collisions between different datacenters. The regional spine, datacenter spine, and leaf devices are all configured to relay default routes received from upstream devices to downstream devices.

Azure's network design assures that all server-server traffic has low latency by coercing the following path conditions: The path between two servers from different clusters in the same datacenter

```
1  VRF name: default
2  Codes: C - connected, S - static, K - kernel,
3         O - OSPF, IA - OSPF inter area, ...
4         B E - eBGP, ...
5         ...
6
7  Gateway of last resort:
8   B E 0.0.0.0/0 [200/0] via 30.10.192.12, ...
9                         via ...
10                        via ...
11                        ....
12
13   B E 10.3.129.224/28 [200/0] via 10.10.192.12, ...
14                               via ...
15                               ...
16
17   '''
```

Figure 2: A routing table from a network device.

comprise a ToR hosting the source server, a leaf switch in the source cluster, a spine switch, a leaf switch in the destination cluster, and finally a ToR hosting the destination server. Similarly, the path between two servers in the same cluster comprise a leaf device in the cluster. These paths assure low latency, and the multiple paths available support maximum bandwidth.

2.2 Forwarding Information Base

The forwarding information base (FIB) of a router determines its forwarding behavior. It is a table, where each entry associates a destination prefix to a set of next hop addresses. Figure 2 contains a routing table from a network device. The network devices use this information to program the FIB, which is maintained in switch application specific integrated circuit (ASIC). Whenever a router receives a packet, it selects the next hop address to forward the packet using two steps. First, it selects the entry with the longest prefix containing the destination address in the packet header. For example, a packet targeting an address matching prefix 10.3.129.224/28 would be processed by the routing rule at line 13. Second, it selects one of next hops from the list of next hops. The routing rule at line

8 is called the *default* route, and is selected when no other rule in the table matches the destination address.

2.3 Reality and Intent

We use the FIB to represent the reality of a device with respect to its forwarding behavior. The forwarding behavior of the entire network is determined by forwarding behavior of each device.

For a datacenter, we postulate intent for forwarding behavior as:

Intent 1. *All pairs ToR reachability.*

Intent 2. *Traffic should always follow a shortest path.*

Intent 3. *All redundant shortest paths should be available.*

The paths between any two ToRs should either be of length 4 or 2. Intra-datacenter paths, i.e., paths between ToRs in different clusters but the same datacenter, must be of length 4. For example, in Figure 1, any path between T0-1 with ASN 65510 and T0-1 with ASN 65410, must consist of a T1 with ASN 65534, T2 with ASN 65535, and T1 with ASN 65533. Intra-cluster paths, i.e., paths between ToRs in the same cluster must be of length 2. For example, any path between T0-1 with ASN 65510 and T0-2 with ASN 65511 must be of length 2. Availability of redundant paths provides maximal flexibility to routing strategies, such as ECMP.

We derive the concrete invariants for each datacenter based on facts about placement of address ranges and topology. We get these facts from a metadata service that maintains these facts.

2.4 Local Validation

A straightforward approach for validating enforcement of intended forwarding policies against the actual network comprises of two steps. First, we need to obtain a stable snapshot of the routing tables from all the devices and form the composite routing table for the entire network. Second, we can then validate the intent against the composite routing table. Obtaining a stable snapshot of the entire network is an engineering feat [50]. Each router in the network may update its routing table using possibly unsynchronized clocks. Merging them in the wrong timestamp order may lead to an inconsistent state that does not occur in practice, and therefore lead to false positives. In contrast, obtaining a stable state from a single switch is deterministic. The complexity of validating the snapshot without additional domain insights is at least cubic in the network graph for all pairs of shortest paths. Furthermore, it exhibits an exponential number of ECMP redundant paths measured by the diameter of the network graph. Even as the diameter is low, 4 in our case, the fan-outs with degree 4-12 produce roughly 1000 different paths per pair of end-points. Validating correctness of the network with hundreds of thousands of ToRs requires verifying a billion routing invariants for all pairs of ToRs.

In bypassing these challenges, we developed a local validation technique. It exploits Azure network's highly regular structure to decompose the end-end reachability invariants into a set of local contracts for every device. Each network device plays a fixed role for a set of address ranges. If the role is enforced in each device, then end-end reachability follows as a consequence. The contracts are expressible as conditions that are local to a device, and the correctness of their enforcement can be validated in each device independently. Therefore, we can parallelize validation and thus scale. Correctness of local forwarding behavior is invariant to small fluctuations in network state, as they are dealt with by routing strategies, obviating the need for obtaining a synchronized snapshot of the entire network.

We illustrate how we decompose the end-end reachability invariants into local contracts using a running example. Figure 3 contains a datacenter topology that is scaled down for the purpose of illustrating contracts, and Figure 4 describes the corresponding local contracts. There are four link failures in Figure 3. We create contracts based on expected topology, and therefore will ignore current state of the links when generating contracts. Subsection 2.4.4 describes how these failures lead to contract violations.

A local forwarding contract for a device consists of a prefix and a set of next hops, and states the expectation that all packets whose destination address matches the given prefix must be forwarded to the specified next hops. Each device has two types of contracts, namely specific and default contracts. A specific contract states the expectation for concrete prefixes. A default contract states the expectation for packets whose destination address does not match the prefixes in any of the specific forwarding rules, and therefore follows a default route. The prefix for this contract is specified as 0.0.0.0/0, however, it is referring to the complement of the union of all prefixes in each of the routing rules.

2.4.1 ToR Contracts. Each ToR has a default contract with next hops set to its neighboring leaf devices. For example, the default contract for ToR_1 specifies $\{A_1, A_2, A_3, A_4\}$ as the next hops.

Each ToR has a specific contract for every prefix hosted in the datacenter besides the prefix that it is configured to announce, and the next hops are set to its neighboring leaf devices. For example, ToR_1 has specific contracts for $Prefix_B$, $Prefix_C$, and $Prefix_D$ with next hops set to $\{A_1, A_2, A_3, A_4\}$.

2.4.2 Leaf Contracts. Each leaf device has a default contract with next hops set to its neighboring spine devices. For example, the default contract for A_1 specifies D_1 as the next hop.

Each leaf device has a specific contract for every prefix hosted in the datacenter. They always forward traffic directly to the ToR devices in the cluster they are covering. For example, A_1 has specific contracts for $Prefix_A$ and $Prefix_B$ with next hops set to ToR_1 and ToR_2 respectively. Other datacenter prefixes are forwarded to spine devices that connect to the leaf devices that connect directly to the prefix. For example, A_2 has a specific route for $Prefix_C$ with next hop set to D_2, which in turn connects to B_2.

2.4.3 Spine Contracts. Each spine device has a default route with next hops set to its neighboring regional spine devices. For example, the default contract for D_1 specifies $\{R_1, R_3\}$ as the next hops.

Each spine device has a specific route for every prefix in the datacenter with the next hops set to its neighboring leaf devices from the cluster hosting the prefix. For example, D_1 has specific contracts for $Prefix_A$ and $Prefix_B$ (prefixes hosted in Cluster A) with next hop set to A_1, which is the only device from Cluster A that connects to D_1. Similarly, D_1 also has specific contracts for $Prefix_C$ and $Prefix_D$ with next hop set to B_1.

2.4.4 Contracts in Action. We illustrate how the link failures in Figure 3 cause violations in the local contracts. ToR_1 has lost

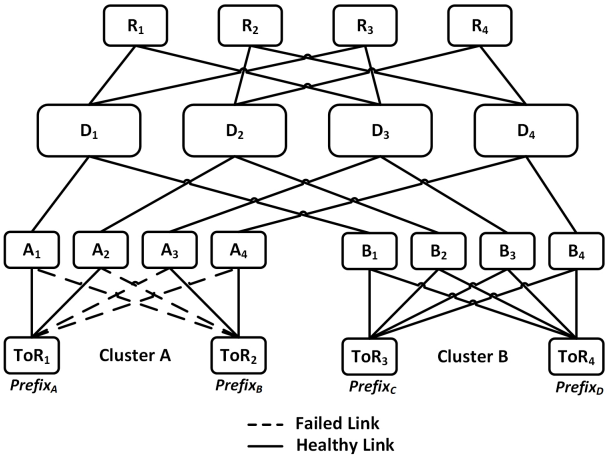

Figure 3: Scaled down topology for illustrating local contracts.

Prefix	Next Hops
0/0	$\{A_1, A_2, A_3, A_4\}$
$Prefix_B$	$\{A_1, A_2, A_3, A_4\}$
$Prefix_C$	$\{A_1, A_2, A_3, A_4\}$
$Prefix_D$	$\{A_1, A_2, A_3, A_4\}$

ToR_1 contracts.

Prefix	Next Hops
0/0	$\{D_1\}$
$Prefix_A$	$\{\text{ToR}_1\}$
$Prefix_B$	$\{\text{ToR}_2\}$
$Prefix_C$	$\{D_1\}$
$Prefix_D$	$\{D_1\}$

A_1 contracts.

Prefix	Next Hops
0/0	$\{R_1, R_3\}$
$Prefix_A$	$\{A_1\}$
$Prefix_B$	$\{A_1\}$
$Prefix_C$	$\{B_1\}$
$Prefix_D$	$\{B_1\}$

D_1 contracts.

Figure 4: Example illustrating local contracts.

its uplinks to devices A_3 and A_4, and ToR_2 has lost its uplinks to devices A_1 and A_2. As a result, ToR_1, A_1, A_2, D_1, and D_2 have a contract failure for $Prefix_B$; these devices do not have a specific route for $Prefix_B$, and packets targeting these destinations choose the default route. ToR_2, A_3, A_4, D_3, and D_4 have a similar failure for $Prefix_A$. Finally, both ToR_1 and ToR_2 have a default contract failure because the default route in both devices have only two next hops compared to the expected four next hops. These failures cause all traffic targeting $Prefix_B$ from ToR_1 to take a longer route. First, such packets must follow default routes all the way up to R_1 or R_2. R_1, R_2, D_3, D_4, A_3, and A_4 have no contract failures for $Prefix_B$. Therefore, the packets must be able to follow the specific routes in those devices to reach ToR_2.

In this example, the default contract failures in both ToR_1 and ToR_2 cause the longer route for traffic from ToR_1 to ToR_2, and vice versa. The absence of default route contract failures in all devices, and specific contracts failures relating to $Prefix_B$ in the R devices ensures the availability of longer routes.

The severity of the failure depends on the number of servers affected by the device, and the number of additional faults required to cause an availability impact. For example, the severity of an error in R_1 is higher than a similar error in D_1. From an operations perspective, it makes sense to address errors in the order of severity. This is particularly important for costly remediations such as those involving manual steps.

2.4.5 *Local Contracts Imply Global Reachability.*

Claim 1. *If local contracts are preserved in the ToR, leaf, and spine devices, then all pairs of ToRs in the datacenter are reachable to one another through the maximal set of shortest paths provided by the redundant routers deployed in the datacenter.*

The local contracts allow us to formulate a global claim about forwarding behavior within a datacenter. In summary, the local contracts on each device ensure that traffic is forwarded along fully redundant routes, e.g. with a fan-out corresponding to the number of redundant routers in each cluster (in Figure 1 called k, n, m, p) in each level. The forwarding behavior also ensures that the shortest paths, which are either of length 2 or 4, are taken.

In the abstract, local validation amounts to checking policies $P_v : H \to 2^{H \times V}$ that at node v map a header $h \in H$ into a set of next nodes and potentially modified headers. It requires a mapping into the natural numbers $\delta : H \times V \to \mathcal{N}$ (perhaps helpful to think of as a *time to live*), such that whenever $(h', v') \in P_v(h)$, then $\delta(h, v) > \delta(h', v')$ and such that when $\delta(h, v) = 0$, then v is the intended destination for header h. It requires a cardinality bound $\mathrm{C} : H \times V \to \mathcal{N}$, that provides lower bounds on number of next hops (for ECMP routing strategies), and has the property $\mathrm{C}(h, v) > 0$ whenever $\delta(h, v) > 0$. It is satisfied by network policies when $|\{v' \mid (h', v') \in P_v(h)\}| \geq \mathrm{C}(h, v)$. Our setting is even more specific than cardinality bounds, it specifies which next hops *must* be reached, it does not involve rewriting headers, and it requires the ordering induced by δ to follow shortest paths. Thus, local validation for forwarding applies if suitable δ and C can be determined across deployed policies.

2.5 Verification Engine

The verification engine takes as input a prefix-based forwarding policy P and a contract C, and produces a list of rules in P that violate the contract. The list is empty if P satisfies C.

By default, the verification engine leverages Z3 [11] by encoding policies and contracts as bit-vector logic formulas, and extracts answers using satisfiability checking. This approach provides a flexible query language and performance is within a second for routing tables extracted from our datacenters. For the most common workload, we developed a specialized and much faster algorithm. It enabled scaling verification to several thousands of devices using very modest CPU resources. The algorithm exploits the fact that address ranges in the contract and routing rules are proper address prefixes (a mask on an IP address that fixes a range of most significant bits). We describe both algorithms.

2.5.1 *Bit-Vector Logic Modeling.*

The predicate for a routing rule that directs packets targeting IP addresses in a prefix 10.20.20.0/24

to next hops A, B, C, or D comprises the following two parts:

$$r_i.prefix(\vec{x}) = (10.20.20.0 \leq \vec{x} \leq 10.20.20.255) \quad (1)$$
$$r_i.nexthops = A \vee B \vee C \vee D \quad (2)$$

The first part of the predicate checks whether the given address $\vec{x}$ is within the address range described by the prefix, The second part of the predicate is disjunction of Boolean variables, each corresponding to a next hop interface.

The meaning of a policy P is defined as a predicate $P(\vec{x})$ that returns the forwarding decision encoded as formula. It is a next hop expression, if a route exists for the destination. Otherwise, it is a Boolean constant *drop* representing that the packet was dropped. Suppose a policy has rules r_1,..,r_n, where the rules in the routing policy are sorted in descending order of the prefix length. Then the meaning of the policy is defined by induction on n:

Definition 2.1 (Longest Prefix Matching Policy). Define P, P_i (for $0 \leq i < n$) and P_n as:

$$P(\vec{x}) = P_1(\vec{x})$$
$$P_i(\vec{x}) = \textbf{if } r_i.prefix(\vec{x}) \textbf{ then } r_i.nexthops \textbf{ else } P_{i+1}(\vec{x})$$
$$P_n(\vec{x}) = drop$$

A contract for a routing policy defines the expectation for forwarding behavior of all packets whose destination address matches an address prefix. The semantics is similar to the routing rule. The outcomes of verifying P using C are either of the following:

(1) $C.range(\vec{x}) \wedge P \wedge \neg C.nexthops$ is unsatisfiable. The forwarding behavior for the prefix stated in the contract is preserved by the policy.
(2) $C.range(\vec{x}) \wedge P \wedge \neg C.nexthops$ is satisfiable: The routing policy selects a different set of next hops in comparison to the behavior expected by the contract.

Several variants of contract checking are possible. For example, we can check whether $C.range(\vec{x}) \wedge \neg(P \Leftrightarrow C.nexthops)$ is satisfiable to enforce that a policy agrees with a contract with respect to all output ports.

Validating a routing contract for the default route, i.e., 0.0.0.0/0 is handled as a special case. Say the policy P has a default route $r_{default}$. For validating a contract $C_{default}$ for the default route, we check that:

$$r_{default}.nexthops = C_{default}.nexthops$$

The intent of validating such a contract is to verify the forwarding behavior of packets that are not handled by any of the routing rules.

2.5.2 Trie-based Algorithm. We represent prefix-based routing policies into a hash-trie such that the shortest prefix match, corresponding to the default route labels the root node of the trie, and such that a node referencing a routing rule r_i is added as a child of a node referencing a routing rule r_j if the prefix of r_i extends r_j, and r_i is not an extension of another rule that also extends r_j.

Validating the default route contract is performed by comparing the next hops in the contract and the default route.

For verifying each contract C, we select the candidate routing rules from the trie as follows:

$$\{r_i \mid (C.range \subseteq r_i.prefix) \vee (r_i.prefix \subseteq C.range)\}$$

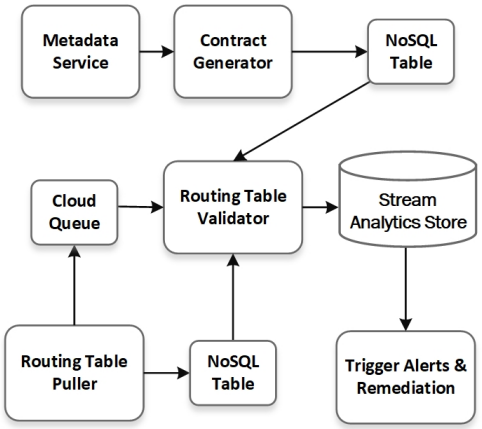

Figure 5: Architecture Diagram

Collecting this set of rules is efficient when $C.range$ is a proper prefix, because traversal of the hash-trie can be limited to nodes that correspond to rules that are returned. We walk through this list in the descending order of prefix length and do the following:

- Check if the *nexthops* match those required by the contract. If they don't, then we add the rule to the list of rules violating the contract.
- Add $r_i.prefix$ to a list L.
- If the $C.range$ is a subset or equals the union of prefixes in L, then we stop. Otherwise, repeat the steps for the next rule.

Finally, we return the list of violating rules.

2.6 Live Monitoring of Forwarding Behavior

We describe a service that we built for continuously verifying the forwarding behavior leveraging local validation. We describe the types of latent bugs we detected, and how we remediated them before they caused an outage.

The Azure datacenter network is designed with adequate redundancies to tolerate a certain number of link or BGP session failures. For example, each top-of-rack switch is connected to m leaf switches and is configured to use equal-cost multi-path (ECMP) routing. Therefore, each switch has m next hops for traffic going in the north direction. Therefore, the top-of-rack switches can tolerate up to $m-1$ failures without being isolated from the network.

It is typical to have a few link failures in a production datacenter. For example, we sometimes mitigate lossy links by shutting down BGP on such links. Only when these failures exceed a threshold we have reachability issues such as longer paths or unreachable destinations. The value of RCDC is in detecting latent bugs, i.e., devices that have already experienced a few failures such that additional failures could cause an outage. Remediating these bugs in a timely fashion will help us avoid outages.

2.6.1 System Architecture. RCDC comprises 3 micro services, namely a device contract generator, a forwarding table puller, and a routing table validator. Figure 5 contains the architecture diagram for the validation service. The device contract generator consumes facts from the metadata system, generates a comprehensive set of

contracts for each device, and pushes them to a NoSQL data store. The routing table puller pulls routing tables from all the network devices periodically, pushes them to an NoSQL data store, and also posts a notification in an cloud queue. The routing table validator consumes each notification, pulls both the routing table and contracts from the NoSQL data store, validates them, and pushes the results to a stream analytics system. Alerts are triggered based on the results.

RCDC is designed for horizontal scalability. The service is partitioned into several instances. Each instance is configured to monitor the devices in a set of datacenters. The service instance, the NoSQL data store, and the queue are chosen to have minimal latency from the set of devices being monitored. Each service instance is configured to monitor O(10K) devices. Fetching each routing table takes 200-800ms, and validating takes O(100) milliseconds. The frequency of validation is configurable.

The stream analytics systems features a query interface that facilitates interactive querying of the results. The alerts and remediations are triggered by a set of queries that correlate the validation errors with additional metadata, classify errors, and direct them appropriately for remediation. For example, if links are operationally down, then these are most likely because of cabling faults and are remediated by replacing the cables. On the other hand, if the BGP sessions are administratively shut, then they are unshut and monitored for health. If they become unhealthy, then they are shut again and directed for further investigation.

2.6.2 Errors. After deploying RCDC, initial reports identified a few hundred latent bugs. Although the absolute number of bugs represent less than one percent of the total number of devices in the network, these still pose a risk to reliability. We describe a set of highly diverse root causes for errors recovered by the RCDC.

Software Bug 1. A software bug in the network device operating system caused a RIB-FIB inconsistency. Those devices used significantly fewer next hops for the default route compared to expected, and therefore violated the default contracts.

Software Bug 2. Another software bug caused the port interfaces on some devices to be treated as layer 2 switch ports instead of layer 3 routed ports, resulting in the device not assigning any IP addresses to the interfaces. As a result, BGP sessions could not be set up on any of the interfaces in those devices, and therefore their routing tables violated all forwarding contracts.

Hardware Failures. Hardware failures on the optical cables caused some links to be operationally down. As a result, BGP sessions on those links were down too; the devices on either end had fewer next hops for traffic targeting the other device violating the local contracts.

Operation Drift. We detected BGP sessions that were shut down administratively to mitigate lossy links, but never remediated. Therefore, all devices on either side of the link violated the routing invariants.

Migrations. Migrating traffic from a decommissioned to new infrastructure comprises multiple steps. Some of these steps may be manual operations, and therefore have high potential for misconfigurations. For example, the leaf devices of both the decommissioned and the new infrastructure were configured with the same ASN numbers. As a result, the top-of-rack switches hosted in both infrastructures did not see the specific route announcements of each other. Therefore, the top-of-rack switches violated all the specific contracts. There were no reachability issues because the traffic between these two clusters were following default routes and reaching the correct destination. However, the lack of specific routes could potentially cause the traffic to use a longer path in the presence of some link failures, and result in higher latency. Therefore, the specific contract violations must be remediated in time.

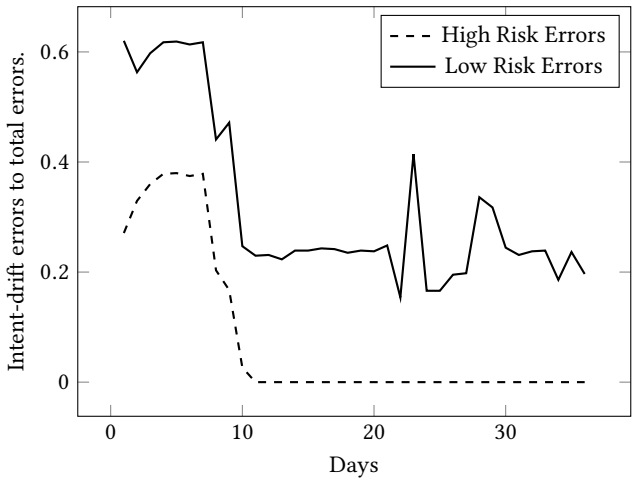

Figure 6: Burndown Graph of Errors.

Policy Errors. We detected several types of policy configuration errors in a part of our network infrastructure that was in part manually configured. We describe two examples. Network devices are configured with policies called route maps that determine the type of BGP announcements that they must accept or send. A policy misconfiguration resulted in devices rejecting default route announcements from upstream devices. Another misconfiguration in the ECMP setup resulted in some of the devices using just a single next hop for upstream traffic instead of all the available upstream devices. The contract violations detected all these issues.

2.6.3 Performance. Most devices in our datacenter network have routing tables with several thousands of prefixes. A topology generator that produces synthetic benchmarks with characteristics similar to the Azure network is available from [29]. Checking a device in a typical Azure datacenter involves checking several thousands of contracts. RCDC takes 180ms to verify all contracts on a single device on average. RCDC can check all-pairs of redundant routes in a datacenter with up to 10^4 routers in less than 3 minutes on a single CPU.

2.6.4 Remediation. Errors are classified by risk factor based on the number of servers it impacts, and the number of additional faults required to cause an impact. For example, a top-of-the-rack switch that has only a single next hop for default route represents a high-risk error, since any additional failure can isolate the top-of-rack switch and cause availability issues for the servers hosted below. Similarly, if a significant number of spine devices in a datacenter have errors relating to specific prefixes, then those errors represent

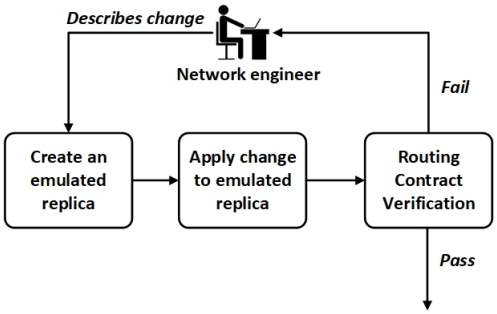

Figure 7: Workflow for validating network changes before they are applied in production.

a high-risk because they are required for assuring the longer paths for several servers. High risk errors are addressed with higher priority, and low risk errors are addressed after.

Validation reports are used to derive automatic alerts, that in turn trigger an automated triaging process. The triaging process collects additional information to direct the error further, determines the risk of the error, and pushes them to an appropriate queue for remediation. For example, errors that requires replacing faulty cables are pushed to a queue maintained for datacenter operations personnel. In all these queues, the high priority errors are remediated before addressing the low-priority errors.

Figure 6 illustrates the observed burndown trend of routing intent-drift errors. The y-axis shows the relative proportion of the high-risk and low-risk errors to total number of errors. It documents a clear downward trend of errors since RCDC was deployed near day 5. It illustrates how the risk assessment helped the DevOps teams prioritize fixing high risk errors quickly.

2.7 Preventing Dangerous Changes

A built-in limitation of live monitoring is that it can only detect dangerous changes after they have occurred. To prevent a large class of faulty updates from entering in the first place Azure uses a high-fidelity network emulator [27]. It runs a full stack of virtualized device software, connected with virtual links using the same topology as the production network, and configured with live configurations and routing state from production devices. To explore more scenarios and scale much further, Azure also uses a network simulator to extract live configurations [31]. RCDC is then used on FIBs extracted from these networks, reporting the same class of errors as on the live network. This pipeline is integral in a *pre-check* process (Figure 7) that is used prior to rolling out configuration changes, whether during a repave or as part of periodic refresh, to production networks. It is used to detect issues arising from root causes such as software bugs, policy errors, and interoperability issues triggered by a specific change.

2.8 Assumptions and Limitations

RCDC assumes a structured datacenter network such as Azure, where the topology and address locality are tracked. We expect our assumptions to hold for datacenter networks based on VL2 [17]. These assumptions enable decomposing the reachability properties to local contracts. Also, RCDC detects changes that affect the steady state routing behavior. In contrast, it may not detect transient errors. For example, it cannot detect a transient routing loop while BGP is still in a transient state.

3 NETWORK CONNECTIVITY RESTRICTIONS

Azure secures its infrastructure and customer services by hosting them in customized isolation boundaries using network connectivity restrictions. For example, Azure management service interfaces are walled off from the Internet and arbitrary customer access. In addition, customer services are also isolated from one another. These restrictions are enforced in network devices such as routers and top-of-rack switches, hypervisor packet filters, and firewalls. Managing these restrictions is fraught with complexity and cannot rely on human inspection or trial and error. This section describes three case studies of deploying static verification techniques for managing these types of policies.

We developed a library called SecGuru for facilitating automatic validation of network connectivity policies using Z3 [11]. The library implements mechanisms for encoding policies and contracts into bit-vector logic, and verifying that the contracts are preserved by the policies. We provide a brief background on the semantics of these policies, describe the design of a verification engine for these policies, and finally describe three case studies that describe our experience of deploying these techniques and the outcome of work in terms of avoiding customer impact.

3.1 Background

This section provides a brief primer on network device access-control lists, network security groups, and their syntax and semantics. Network devices can be configured with an access-control list (ACL) to enforce restrictions on traffic flowing through the device. Figure 8 contains an example ACL. This ACL is authored in Cisco's IOS language, used by most Cisco devices. Network devices from other vendors use similar languages. Network security groups (NSG) help customers enforce connectivity restrictions on resources deployed inside their virtual network (Figure 9). All IP addresses used in Figures 8 & 9 are for illustrative purposes only.

The syntax of the two policies vary, but semantics is similar. In both cases, a policy is a set of rules. Each rule describes a packet filter and an action. The packet filter describes permissible values for source and destination addresses, source and destination ports, and protocol. The expression 10.0.0.0/8 specifies an address range 10.0.0.0 to 10.255.255.255. That is, the first 8 bits are fixed and the remaining 24 (= 32-8) vary. A wild card is indicated by *Any*. For ports, *Any* encodes the range from 0 to $2^{16} - 1$. The action is either *Permit* or *Deny*. They indicate whether packets matching the range should be allowed through the firewall.

Both policies have the first-applicable rule semantics, where the device processes an incoming packet per the first rule that matches its description. If no rules match, then the incoming packet is denied by default. Therefore, the order in which the rules appear is important. For ACL, the order is implicit in the sequence of rules. For NSG, the priority field specifies the order: smaller numbers have higher priority.

```
1   remark Isolating private addresses
2   deny ip 0.0.0.0/32 any
3   deny ip 10.0.0.0/8 any
4   deny ip 172.16.0.0/12 any
5   deny ip 192.0.2.0/24 any
6   ...
7   remark Anti spoofing ACLs
8   deny ip 104.208.32.0/20 any
9   deny ip 168.61.144.0/20 any
10  ...
11  remark permits for IPs without
12         port and protocol blocks
13  permit ip any 104.208.32.0/24
14  ....
15  remark standard port and protocol
16         blocks
17  deny   tcp any any eq 445
18  deny   udp any any eq 445
19  deny   tcp any any eq 593
20  deny   udp any any eq 593
21  ...
22  deny   53 any any
23  deny   55 any any
24  ...
25  remark permits for IPs with
26         port and protocol blocks
27  permit ip any 104.208.32.0/20
28  permit ip any 168.61.144.0/20
29  ...
```

Figure 8: Network device access-control list.

```
1   [{
2     "Name": "Allow Prefix 1",
3     "Id": "Prefix2",
4     "Etag": "45aa-c152",
5     "Properties": {
6       "ProvisioningState": "Succeeded",
7       "Protocol": "TCP",
8       "SourcePortRange": "1024-65535;500",
9       "SourceAddressPrefix": "SQLMI_TAG;40.79.210.20/32;
10                             20.190.140.128/25",
11      "DestinationPortRange": "1433",
12      "DestinationAddressPrefix": "Any",
13      "Action": "Allow",
14      "Priority": 6000,
15      "Direction": "Inbound" }
16  },
17  {
18    "Name": "Deny override",
19    "Id": "Prefix2",
20    "Etag": "45aa-c152",
21    "Properties": {
22      "ProvisioningState": "Succeeded",
23      "Protocol": "Any",
24      "SourcePortRange": "Any",
25      "SourceAddressPrefix": "Any",
26      "DestinationPortRange": "Any",
27      "DestinationAddressPrefix": "Any",
28      "Action": "Deny",
29      "Priority": 100,
30      "Direction": "Inbound" }
31  }]
```

Figure 9: Network security groups.

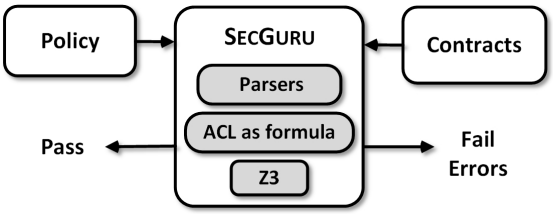

Figure 10: SecGuru architecture.

The ACL in Figure 8 has five sections: §1, lines 2-6, filter traffic that targets private IP addresses. For example, line 3 blocks traffic targeting 10.0.0.0/8, which is a private address range per RFC1918; §2, lines 8-10, are for anti-spoofing; §3, lines 13-14, permit traffic targeting IP addresses that should be permissible without any port blocks; §4, lines 17-24, block a standard set of ports and protocols on all Internet traffic targeting any destination inside the network; §5, lines 27-29, permit traffic to a set of address ranges.

3.2 Verification Engine

SecGuru takes a policy and a set of contracts as input, and produces the outcome of analysis as output (Figure 10). In the case of network device ACLs, the policy is the configuration of the network device and the name of the ACL that it contains and needs to be analyzed. In the case of NSGs, the policy is a file containing the NSG rules. Each contract, similar to a policy rule, describes a packet filter and expectation of whether the packets matching the description must be permitted or denied. SecGuru encodes policies and contracts as predicates in bit-vector logic, and leverages satisfiability checking to extract answers. The intuition behind this design is that network connectivity restrictions are essentially a set of constraints over IP addresses, ports, and protocol, each of which are bit-vectors of varying sizes. Modeling policy analysis questions as logical formulas allows analysis to be semantic and agnostic to the low-level device syntax used for access control. A tool derived from how SecGuru encodes and checks ACLs, and specialized to Windows Firewalls, is open source [20].

The outcome of SecGuru analyzing a policy, (P), and a contract, (C), is one of the following:

(1) $C \rightarrow P$ is valid: The contract is preserved by the policy, i.e., the set of all traffic patterns described by C is a subset of the set of all traffic patterns accepted by the policy.
(2) $C \wedge \neg P$ is satisfiable: The contract is not preserved by the policy, i.e., some traffic patterns accepted by C are denied by P. In this case, the error report also identifies the rule in the policy that violated the contract.

We describe how policies and contracts are encoded as predicates using an example. The rules corresponding to lines 3 and 13 of Figure 8 have the associated predicates:

$$r_3: \quad (10.0.0.0 \leq srcIp \leq 10.255.255.255) \wedge protocol = 4$$

$$r_{13}: \quad (104.208.32.0 \leq dstIp \leq 104.208.32.255) \wedge protocol = 4$$

We use $r_i(\vec{x})$ to refer to the predicate associated with the i'th rule in a policy. The tuple $\vec{x}$ abbreviates

$$\langle srcIp, srcPort, dstIp, dstPort, protocol \rangle.$$

We use $r.Action$ to access the action field of a rule. It is either *Allow* or *Deny*.

The meaning of a policy P is defined as a predicate $P(\vec{x})$ that evaluates to *true* when a packet with header $\vec{x}$ is allowed to pass through. There are two conventions for combining the rules in a policy, namely *Deny Overrides* and *First Applicable*. We describe the semantics of policies according to these two conventions.

Network device ACLs and NSGs use the first applicable rule semantics. Suppose an ACL has rules $r_1, \ldots, r_n$ that are either Allow or Deny rules, then the meaning is defined (linear in the size of the policy) by induction on n:

Definition 3.1 (First Applicable Policies). Define P, P_i (for $0 \leq i < n$) and P_n as:

$$\begin{aligned} P(\vec{x}) &= P_1(\vec{x}) \\ P_i(\vec{x}) &= r_i(\vec{x}) \vee P_{i+1}(\vec{x}) \quad \textbf{if } r_i.action = Allow \\ P_i(\vec{x}) &= \neg r_i(\vec{x}) \wedge P_{i+1}(\vec{x}) \quad \textbf{if } r_i.action = Deny \\ P_n(\vec{x}) &= false \end{aligned}$$

Definition 3.2 (Deny Overrides Policies). Let $Allow = \{r \in P \mid r.action = Allow\}$ and likewise $Deny = \{r \in P \mid r.status = Deny\}$. The meaning of P with the Deny Overrides convention is the formula (linear in the size of the policy):

$$P(\vec{x}) = \Big(\bigvee_{r \in Allow} r(\vec{x})\Big) \wedge \Big(\bigwedge_{r \in Deny} \neg r(\vec{x})\Big)$$

Thus, a packet is admitted if some Allow rule applies and none of the Deny rules apply.

We showed how policies correspond to predicates over bit-vectors. Both policies using the Deny Overrides and the First Applicable semantics correspond to logical formulas. The predicates treat the parameters as bit-vectors and use comparison (less than, greater-than, equals) operations on the bit-vectors as unsigned numbers. Modern SMT, Satisfiability Modulo Theories, solvers contain efficient decision procedures for bit-vector logic. Bit-vector logic is expressive: it captures the operations that are common on machine represented fixed-precision integers, such as modular addition, subtraction, multiplication, bit-wise logical operations, and comparisons. The solvers leverage pre-processing simplifications at the level of bit-vectors and most solvers reduce formulas to propositional satisfiability where state-of-the-art SAT solving engines are used. In the worst case the underlying solver could use an algorithm that is asymptotically much worse than algorithms that have been specifically tuned to policy analysis (as for instance developed in [1, 5]). However, our experience shows that our approach easily scales to an order of magnitude beyond what is required for modern datacenters. For example, analyzing an ACL comprising a few hundred rules takes approximately 300ms and analyzing an ACL comprising a few thousand rules takes a second.

3.3 Managing Legacy Policies

Azure deploys an ACL in all network devices that peer with Internet service providers. The structure of this ACL is similar to the ACL described in Figure 8. It is called an Edge ACL, and had inorganically grown to comprise several thousand rules. For example, for every new prefix that Azure acquired, we needed planned updates to §2, and §3 or §5. In addition to the sections in Figure 8, there were also several service specific rules. For example, several services enforced a whitelist of addresses for their instances in the Edge ACL. There were also several deny rules interspersed at several places in the ACL to mitigate zero day attacks. All the changes together resulted in the ACL growing to several thousand lines. The semantics and the size together made it difficult for engineers to assess the impact of changes to the ACL manually. As a result, we had a few availability issues from misconfiguring the ACL.

We had to refactor the ACL to make it more manageable. The intent for the new ACL was to only enforce private address isolation, anti-spoofing protection, and protections common for all services.

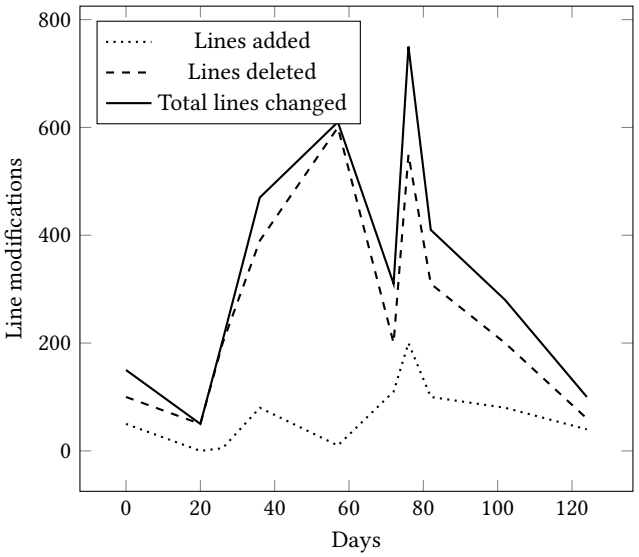

Figure 11: Managing the complexity of a legacy ACL.

Service specific protections would be moved to end-host mechanisms such as host firewalls. The scope of the change was huge because the ACL impacts nearly every service in Azure. A simple typo could potentially cause several services to be inaccessible. We needed a methodology for safely transitioning the current ACL to the new ACL without causing any negative impact for any service.

Our methodology was to design a phased plan for refactoring the ACL. The idea was to incrementally transform the ACL to the expected goal state. We designed each change to consist of a set of prechecks, the change, postchecks, and finally a rollback methodology if the postchecks fail. Prechecks are to ensure that the change, if performed on the actual device, would result in an new ACL that preserves the intent. Failing prechecks must provide information to help fix the error in the change. Postchecks are to ensure that the change leaves the production device in the expected state. The production devices are partitioned into distinct groups, and the change is deployed in one group at a time. Successful postchecks in a group are a precondition for deploying the change in the next group. Deploying the change in groups helps in limiting the impact from misconfigurations to a subset of services. For example, partitions can be designed based on devices supporting a particular region.

We leveraged SecGuru to perform prechecks and postchecks. SecGuru consumes the device configuration extracted from the device, the name of the ACL, and a set of contracts. Each contract expressed a reachability invariant such as private datacenter addresses must not be reachable from the Internet or a list of services that must be reachable on port 80 and 443 from the Internet. These contracts are essentially a set of regression tests for the ACL. The prechecks were designed to be executed on a test network device. The test network device was first configured with the existing ACL and then re-configured with the new ACL. The new configuration that incorporated the modified ACL was then fed as input to SecGuru along with the contracts. This methodology allowed us to

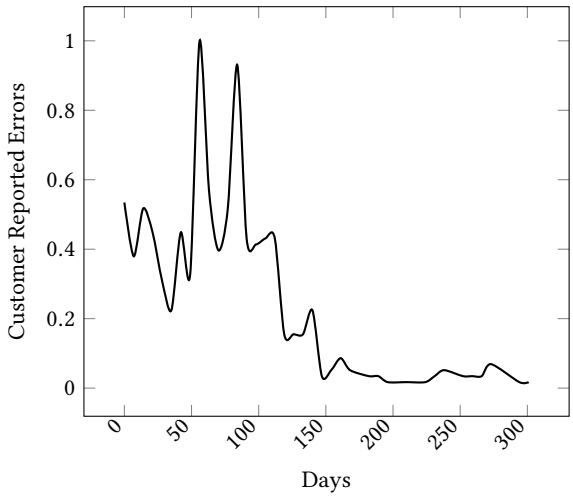

Figure 12: Burndown graph of customer issues.

validate that the effective ACL after making the change preserves all the contracts. For example, if resource limitations on the device cause certain additional rules to be ignored, then the effective ACL in the configuration would violate the contracts. If all the prechecks pass, then the change was deployed on the production device. With each refactoring step, we added additional contracts to cover the most recent updates. We deployed SecGuru in our production monitoring service to perform the checks on every production device on demand. These became our post-checks.

Figure 11 describes the changes that we performed on the ACL. Each change incrementally deleted several rules that were either unnecessary or redundant, and also added new rules as necessary. The prechecks we deployed helped us identify several misconfigurations when making these changes. For example, pre-checks detected typos, such as incorrect prefixes, that caused several services to be unreachable. In the absence of prechecks, these changes would have caused an outage. In the end, we were able to reduce the ACL to less than 1000 lines without outages or business impact.

3.4 Safeguarding Network Security Groups

In a recent experience, Azure released a new cloud-based fully managed database solution. In an on-premises network, customers would deploy and manage their own private database services and the applications that interact with them. Customers would deploy their applications in virtual networks and purchase a managed service instance of the database that would be automatically deployed in their virtual networks. A distinguishing feature of the managed database is that it would periodically backup data. The backup is initiated and orchestrated by an infrastructure service that is not part of the virtual network. Therefore, a critical requirement is that the database instance must be able to reach the infrastructure service. A common problem was that customers were inadvertently misconfiguring the NSGs applied to their virtual networks and blocking access to the infrastructure service. As a result, the periodic backups were failing. All such issues reported by customers were diagnosed to NSG policies that were blocking access.

Customers who were making changes to the NSG policies were not aware that they were blocking database backups. The first applicable semantics of NSGs made it hard for operators to reason about such policies comprising several hundred rules. For some enterprise customers, different teams managed the NSG policies and database services, and the team changing the NSG policies didn't know that they were blocking database backup. Thus, a customer requirement emerged to prevent changes to NSG policies that block database backups. Azure infrastructure has access to metadata about all service addresses and whether the virtual network of a customer included a database instance. Both these facts could be leveraged for blocking unsafe changes to the NSG policies, and avoid database backups from failing.

We extended SecGuru to analyze NSG policies. The analysis engine would take as input a NSG policy and a set of reachability invariants, and produce as output a status report with errors. The report contains a list of invariants that failed, and for each invariant that failed the specific rule in the NSG that caused the failure is also enumerated. The list is empty if all invariants pass. We integrated SecGuru validation into the API for changing NSG policies. We designed service infrastructure to automatically add contracts for ensuring reachability of the database instance with infrastructure services. The API was designed to validate these contracts against the new policy and fail with an error message if the new policy could block database backups.

Figure 12 illustrates the rising trend of customer reported issues relating to misconfigured NSG policies that blocked reachability between the database instance and infrastructure service. When the managed database instance service was initially launched, we saw a steep increase in customer reported issues; since incorporating SecGuru into the validation API, we observed a steep decrease in such customer reported issues (around day 100 in the graph). Fluctuations in the number of reported incidents are based on customer volume and the adoption rate of the NSG checker.

3.5 Validating Distributed Firewalls

Azure enforces a common set of restrictions for every virtual machine. These restrictions ensure that guest virtual machines have no access to infrastructure services, and are also isolated from one another. These common restrictions are specified using a configuration file and are automatically derived from a template. A problem we encountered in the past is that bugs in the automation or policy changes have resulted in restrictions being omitted in deployments.

We extracted a set of contracts that specify our security policy for the common restrictions. The firewall policies described in the configuration file follow the *deny overrides* semantics. We extended SecGuru to model these policies, and incorporated the checking of policies in automation that gates deployments of policies to only those that pass validation. Incorporating validation as part of the deployment process eradicated the previous case when restrictions would accidentally be omitted.

3.6 Assumptions and Limitations

The functionality of SecGuru is generic to network access restrictions. Variants of SecGuru have since 2013 been developed independently by network operators and researchers outside of

Azure. Our use cases for SecGuru did so far not involve checking combinations of firewall policies across devices. Potentially interesting scenarios, such as checking customer virtual networks in context of routing rules are simple extensions, but not in scope for this paper.

4 RELATED WORK

Checking forwarding and network access restrictions has long been a central theme in engineering networks in datacenters, corporate, campus and wide-area networks.

The seminal work in [46] introduces methodologies for statically checking reachability properties in IP networks. Assuming that IP forwarding rules induce a small graph, the different modalities, may and must-reachability, are solved using graph algorithms. As forwarding rules are typically formulated using masks on IP headers, tools based on symbolic SAT solving were proposed in [32] such that a potentially exponential number of forwarding behaviors could be analyzed. To address scalability and performance, this work was followed by tailored data-structures and algorithms [22, 23, 30] and with further optimizations to solve reachability properties under incremental network updates [21, 50]. These approaches handle incremental updates efficiently as they compute the effect of updates within smaller slices. A number of commercial products for global reachability checks have emerged from [13, 22, 23]. Related to these approaches, the Tiros system at AWS [14] addresses customer network configurations by applying both SAT and Datalog engines for global reachability checks.

A complementary, related, line of work considers consistent incremental updates to networks, such that the transition from one network forwarding state to another is transactional [9, 35, 42]. Meanwhile, [47] made a central observation that forwarding behavior can often be partitioned into relatively few equivalence classes, such that IP headers within each equivalence class enjoy the same forwarding behavior. This insight enabled another perspective on handling forwarding in IP networks by pre-processing, and then relying on methods in line with [46] for checking reachability properties. Further coalescing of forwarding behavior was shown to be possible by considering symmetries and bisimulations in datacenter networks [41] and in control plane configurations [4]. A common assumption in all these approaches is the quest for tools that can automatically *discover* network structure and perform global network analysis on networks. Our starting point, on the other hand, is a fixed datacenter design, where routers are classified to have fixed roles and forwarding behavior follows pre-defined patterns, even in the presence of faults.

Early tools for checking network access restrictions applied simulation, such as the commercial firewall analyzer AlgoSec [2] based on Fang [34] and its successor Lumeta [45]. They let administrators answer queries involving router and firewall configurations. Extending simulation, [26] proposes a structured firewall query language, supported by trie-based data-structures. Subsequent tools also used specialized algorithms and data-structures, such as Vantage [5, 6, 16, 18, 33] corresponding to variants of binary decision diagrams [8] or intervals. Tools that use declarative encodings and logic-based solvers have emerged in the past decade and found their way into startups and the major cloud providers. Declarative methods were used in [36, 37], first using logic programming tools, and in subsequent work using SMT solvers. The Margrave firewall analysis engine [38] encodes firewall rules and queries into first-order logic and uses the KodKod relational algebra solver based on propositional SAT technologies. Firewall rules are encoded into integer linear arithmetic available in SMT solvers in [48]. Recent advances involves SAT and SMT based logic engines to synthesize or repair firewall policies and even control plane policies [7, 10, 12, 19, 40, 51]. A key distinction of our work is that we automatically extract intent based on architecture and metadata, and incorporate checking into change processes and deployment workflows.

5 CONCLUSION

This paper addresses dataplane verification in large datacenters and is used as an integral part of configuration and monitoring. The journey that led us to our approach started in an era where datacenter designs were still evolving, and of-the-shelf tools for verifying configurations were in an infancy. The experiences described in this paper were part of the journey: as new services were added, new failure scenarios were revealed and in the cases documented here mapped to validation with RCDC or SecGuru. A central question included *what is the network intent and how is it captured?*. Once addressed, the operational aspect becomes important: *where does verification fit into the DevOps pipeline: with pre-checks, as part of live monitoring, or as a tool for evolving legacy policies?* The final component of our quest is flexibility and scalability by using efficient theorem provers that work over formalisms that are easy to map into from configurations and specifications, and identifying structural properties that enable local scalable techniques that tackle the main queries of interest in the datacenters. We claim that our approaches transfer beyond the setting described in this paper, but note that the underlying claim for the effectiveness of SecGuru and RCDC relies on the assumption that important and even system-wide correctness properties can be checked locally by inspecting device configurations in isolation. For the viability of local methods we assumed that network architectures remain mostly fixed, while network state could change at variable rates. Several opportunities for checking network configurations abound. For example, an extension to WAN properties could catch issues with peering, but checking the WAN is riddled with new challenges: routing tables are order of magnitudes larger than for datacenters impeding downloads and analysis and relevant intent is different.

The quest for global network verification dominance continues.

ACKNOWLEDGMENTS

We would like to thank the reviewers, our shepherd Minlan Yu, Ryan Beckett, and Kevin Schoonover for their very constructive and useful feedback that helped improve the paper in several ways. We would also like to thank Monica Machado who provided both initial inspiration for the design of RCDC and involvement in designing the datacenter contracts; Charlie Kaufman and Geoff Outhred for their influence on the design of SecGuru; Sreenivas Addagatla for architectural guidance; and for numerous interactions with Nuno Lopes and Andrey Rybalchenko around validation of configurations produced by the Network Logic Solver.

REFERENCES

[1] Hrishikesh B. Acharya and Mohamed G. Gouda. 2009. Linear-Time Verification of Firewalls. In *ICNP*. 133–140.

[2] Algorithmic Security Inc. 2006. Firewall Analyzer: Make your firewall really safe. www.algosec.com (Whitepaper).

[3] Shrutarshi Basu, Nate Foster, Hossein Hojjat, Paparao Palacharla, Christian Skalka, and Xi Wang. 2017. Life on the Edge: Unraveling Policies into Configurations. In *ACM/IEEE Symposium on Architectures for Networking and Communications Systems, ANCS 2017, Beijing, China, May 18-19, 2017*. 178–190.

[4] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. 2018. Control plane compression. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication, SIGCOMM 2018, Budapest, Hungary, August 20-25, 2018*. 476–489.

[5] Sandeep Bhatt, Cat Okita, and Prasad Rao. 2008. Fast, Cheap, and in Control: Towards Pain-Free Security. In *USENIX Systems Administration Conference*. 75–90.

[6] Sandeep Bhatt and Prasad Rao. 2007. *Enhancements to the Vantage Firewall Analyzer.* Technical Report HPL-2007-154R1. HP Laboratories.

[7] Chiara Bodei, Pierpaolo Degano, Letterio Galletta, Riccardo Focardi, Mauro Tempesta, and Lorenzo Veronese. 2018. Language-Independent Synthesis of Firewall Policies. In *2018 IEEE European Symposium on Security and Privacy, EuroS&P 2018, London, United Kingdom, April 24-26, 2018*. 92–106.

[8] Randal E. Bryant. 1992. Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams. *ACM Comput. Surv.* 24, 3 (1992), 293–318.

[9] Pavol Cerný, Nate Foster, Nilesh Jagnik, and Jedidiah McClurg. 2016. Optimal Consistent Network Updates in Polynomial Time. In *Distributed Computing - 30th International Symposium, DISC 2016, Paris, France, September 27-29, 2016. Proceedings*. 114–128.

[10] Haoxian Chen, Anduo Wang, and Boon Thau Loo. 2018. Towards Example-Guided Network Synthesis. In *Proceedings of the 2nd Asia-Pacific Workshop on Networking, APNet 2018, Beijing, China, August 02-03, 2018*. 65–71.

[11] L. de Moura and N. Bjørner. 2008. Z3: An Efficient SMT Solver. In *TACAS 08*.

[12] Ahmed El-Hassany, Petar Tsankov, Laurent Vanbever, and Martin T. Vechev. 2017. Network-Wide Configuration Synthesis. In *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II*. 261–281.

[13] Ari Fogel, Stanley Fung, Luis Pedrosa, Meg Walraed-Sullivan, Ramesh Govindan, Ratul Mahajan, and Todd D. Millstein. 2015. A General Approach to Network Configuration Analysis. In *12th USENIX Symposium on Networked Systems Design and Implementation, NSDI 15, Oakland, CA, USA, May 4-6, 2015*. USENIX Association, 469–483. https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/fogel

[14] Andrew Gacek, John Backes, Byron Cook, Neha Rungta, Sam Bayless, Catherine Dodge, Carsten Varming, Alan Hu, Bill Kocik, Evgenii Kotelnikov, Jure Kukovec, Sean McLaughlin, Jason Reed, John Sizemore, Mark Stalzer, Preethi Srinivasan, Pavle Subotic, Blake Whaley, Yiwen Wu, and Temesghen Kahsai. 2019. Reachability Analysis for AWS-based Networks. In *Computer Aided Verification - 31st International Conference, CAV*.

[15] Sergey Gorinsky and János Tapolcai (Eds.). 2018. *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication, SIGCOMM 2018, Budapest, Hungary, August 20-25, 2018*. ACM. http://dl.acm.org/citation.cfm?id=3230543

[16] Mohamed G. Gouda and Alex X. Liu. 2007. Structured firewall design. *Computer Networks* 51, 4 (2007), 1106–1120.

[17] Albert Greenberg, James R. Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A. Maltz, Parveen Patel, and Sudipta Sengupta. 2009. VL2: A Scalable and Flexible Data Center Network. In *Proceedings of the ACM SIGCOMM 2009 Conference on Data Communication (SIGCOMM '09)*. ACM, New York, NY, USA, 51–62.

[18] Swati Gupta, Kristen LeFevre, and Atul Prakash. 2009. SPAN: a unified framework and toolkit for querying heterogeneous access policies. In *HotSec*. USENIX, 5–5.

[19] William T. Hallahan, Ennan Zhai, and Ruzica Piskac. 2017. Automated repair by example for firewalls. In *2017 Formal Methods in Computer Aided Design, FMCAD 2017, Vienna, Austria, October 2-6, 2017*. 220–229.

[20] Andrew Helwer. 2018. Z3Prover/FirewallChecker. https://github.com/Z3Prover/FirewallChecker

[21] Alex Horn, Ali Kheradmand, and Mukul R. Prasad. 2017. Delta-net: Real-time Network Verification Using Atoms. In *14th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2017, Boston, MA, USA, March 27-29, 2017*. 735–749.

[22] Peyman Kazemian, Michael Chan, Hongyi Zeng, George Varghese, Nick McKeown, and Scott Whyte. 2013. Real Time Network Policy Checking Using Header Space Analysis. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2013, Lombard, IL, USA, April 2-5, 2013*. 99–111.

[23] Ahmed Khurshid, Xuan Zou, Wenxuan Zhou, Matthew Caesar, and P. Brighten Godfrey. 2013. VeriFlow: Verifying Network-Wide Invariants in Real Time. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. USENIX, Lombard, IL, 15–27. https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/khurshid

[24] Petr Lapukhov, Ariff Premji, and Jon Mitchell. 2016. Use of BGP for Routing in Large-Scale Data Centers. RFC 7938. https://doi.org/10.17487/RFC7938

[25] Andrew Lerner. 2017. Intent-based Networking. https://blogs.gartner.com/andrew-lerner/2017/02/07/intent-based-networking

[26] Alex X. Liu, Mohamed G. Gouda, Huibo H. Ma, and Anne H. H. Ngu. 2004. Firewall Queries. In *International Conference On Principles Of DIstributed Systems*. 197–212.

[27] Hongqiang Harry Liu, Yibo Zhu, Jitu Padhye, Jiaxin Cao, Sri Tallapragada, Nuno P. Lopes, Andrey Rybalchenko, Guohan Lu, and Lihua Yuan. 2017. CrystalNet: Faithfully Emulating Large Production Networks. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*. ACM, 599–613. https://doi.org/10.1145/3132747.3132759

[28] Jed Liu, William Hallahan, Cole Schlesinger, Milad Sharif, Jeongkeun Lee, Robert Soulé, Han Wang, Calin Cascaval, Nick McKeown, and Nate Foster. 2018. p4v: practical verification for programmable data planes, See [15], 490–503. https://doi.org/10.1145/3230543.3230582

[29] Nuno Lopes. [n. d.]. Cloud Topology Generator. http://web.ist.utl.pt/nuno.lopes/netverif/netverif-scripts-0.2.tar.bz2

[30] Nuno P. Lopes, Nikolaj Bjørner, Patrice Godefroid, Karthick Jayaraman, and George Varghese. 2015. Checking Beliefs in Dynamic Networks. In *12th USENIX Symposium on Networked Systems Design and Implementation, NSDI 15, Oakland, CA, USA, May 4-6, 2015*. 499–512.

[31] Nuno P. Lopes and Andrey Rybalchenko. 2019. Fast BGP Simulation of Large Datacenters. In *Verification, Model Checking, and Abstract Interpretation - 20th International Conference, VMCAI 2019, Cascais, Portugal, January 13-15, 2019, Proceedings (Lecture Notes in Computer Science)*, Constantin Enea and Ruzica Piskac (Eds.), Vol. 11388. Springer, 386–408. https://doi.org/10.1007/978-3-030-11245-5_18

[32] Haohui Mai, Ahmed Khurshid, Rachit Agarwal, Matthew Caesar, Brighten Godfrey, and Samuel Talmadge King. 2011. Debugging the data plane with anteater. In *Proceedings of the ACM SIGCOMM 2011 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, Toronto, ON, Canada, August 15-19, 2011*. 290–301.

[33] Robert M. Marmorstein and Phil Kearns. 2005. An Open Source Solution for Testing NAT'd and Nested iptables Firewalls. In *LISA*. 103–112.

[34] Alain J. Mayer, Avishai Wool, and Elisha Ziskind. 2000. Fang: A Firewall Analysis Engine. In *IEEE Symposium on Security and Privacy*. 177–187.

[35] Jedidiah McClurg, Hossein Hojjat, Pavol Cerný, and Nate Foster. 2015. Efficient synthesis of network updates. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*. 196–207.

[36] Sanjai Narain, Gary Levin, Sharad Malik, and Vikram Kaul. 2008. Declarative Infrastructure Configuration Synthesis and Debugging. *J. Netw. Syst. Manage.* 16, 3 (Sept. 2008), 235–258.

[37] Sanjai Narain, Rajesh Talpade, and Gary Levin. 2009. Network Configuration Validation. In *Guide to Reliable Internet Services and Application*, Charles Kalmanek, Richard Yang, and Sudip Misra (Eds.).

[38] Timothy Nelson, Christopher Barratt, Daniel J. Dougherty, Kathi Fisler, and Shriram Krishnamurthi. 2010. The margrave tool for firewall analysis. In *LISA*. USENIX Association, Berkeley, CA, USA, 1–8.

[39] Aurojit Panda, Ori Lahav, Katerina J. Argyraki, Mooly Sagiv, and Scott Shenker. 2017. Verifying Reachability in Networks with Mutable Datapaths. In *14th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2017, Boston, MA, USA, March 27-29, 2017*, Aditya Akella and Jon Howell (Eds.). USENIX Association, 699–718. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/panda-mutable-datapaths

[40] Ruzica Piskac. 2018. New Applications of Software Synthesis: Verification of Configuration Files and Firewall Repair. In *Static Analysis - 25th International Symposium, SAS 2018, Freiburg, Germany, August 29-31, 2018, Proceedings*. 71–76.

[41] Gordon D. Plotkin, Nikolaj Bjørner, Nuno P. Lopes, Andrey Rybalchenko, and George Varghese. 2016. Scaling network verification using symmetry and surgery. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*. 69–83.

[42] Mark Reitblatt, Nate Foster, Jennifer Rexford, and David Walker. 2011. Consistent updates for software-defined networks: change you can believe in!. In *Tenth ACM Workshop on Hot Topics in Networks (HotNets-X), HOTNETS '11, Cambridge, MA, USA - November 14 - 15, 2011*. 7.

[43] Radu Stoenescu, Dragos Dumitrescu, Matei Popovici, Lorina Negreanu, and Costin Raiciu. 2018. Debugging P4 programs with vera, See [15], 518–532. https://doi.org/10.1145/3230543.3230548

[44] George Varghese. 2015. Technical Perspective: Treating Networks Like Programs. *Commun. ACM* 58, 11 (Oct. 2015), 112–112. https://doi.org/10.1145/2823394

[45] Avishai Wool. 2001. Architecting the Lumeta Firewall Analyzer. In *Proceedings of the 10th Conference on USENIX Security Symposium - Volume 10 (SSYM'01)*. USENIX Association, Berkeley, CA, USA, Article 7.

[46] Geoffrey G. Xie, Jibin Zhan, David A. Maltz, Hui Zhang, Albert G. Greenberg, Gísli Hjálmtýsson, and Jennifer Rexford. 2005. On static reachability analysis of IP networks. In *INFOCOM 2005. 24th Annual Joint Conference of the IEEE Computer and Communications Societies, 13-17 March 2005, Miami, FL, USA*. 2170–2183.

[47] Hongkun Yang and Simon S. Lam. 2016. Real-Time Verification of Network Properties Using Atomic Predicates. *IEEE/ACM Trans. Netw.* 24, 2 (2016), 887–900.

[48] N. Ben Souayeh Ben Youssef and Adel Bouhoula. 2010. Automatic Conformance Verification of Distributed Firewalls to Security Requirements. In *IEEE ICSC*. 834–841.

[49] Arseniy Zaostrovnykh, Solal Pirelli, Luis Pedrosa, Katerina J. Argyraki, and George Candea. 2017. A Formally Verified NAT. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication, SIGCOMM 2017, Los Angeles, CA, USA, August 21-25, 2017*. 141–154.

[50] Hongyi Zeng, Shidong Zhang, Fei Ye, Vimalkumar Jeyakumar, Mickey Ju, Junda Liu, Nick McKeown, and Amin Vahdat. 2014. Libra: Divide and Conquer to Verify Forwarding Tables in Huge Networks. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. USENIX Association, Seattle, WA, 87–99. https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/zeng

[51] Shuyuan Zhang, Abdulrahman Mahmoud, Sharad Malik, and Sanjai Narain. 2012. Verification and synthesis of firewalls using SAT and QBF. In *ICNP*. 1–6.