Putting network verification to good use

Ryan Beckett
Microsoft Research

Ratul Mahajan
University of Washington
Intentionet

ABSTRACT

The past decade has witnessed remarkable progress in the field of network verification, and interest from academia and industry has spurred the development of increasingly sophisticated verification tools and algorithms. However, outside of a handful of large cloud computing providers, the use of network verification is still sparse. We argue that the next frontier for network verification is to enable easy and effective use by "average" network engineers. Whereas in software development, practitioners frequently use testing frameworks to describe the expected behavior of their systems and to measure the effectiveness of their tests through metrics such as code coverage, no such frameworks exist for the equally challenging task of designing and maintaining networks. To address this gap, we outline the design of a network verification framework. In doing so, we propose 1) a method to compute test coverage for networks, which tells engineers how well their invariants are testing the network; and 2) a new declarative invariant language that makes it easy to express network invariants and enables computation of coverage metrics.

CCS CONCEPTS

• **Networks** → *Network reliability*; *Network manageability*;

KEYWORDS

network verification, testing, code coverage

ACM Reference Format:
Ryan Beckett and Ratul Mahajan. 2019. Putting network verification to good use. In *ACM Workshop on Hot Topics in Networks (HotNets '19), November 13–15, 2019, Princeton, NJ, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3365609.3365866

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
HotNets '19, November 13–15, 2019, Princeton, NJ, USA
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-7020-2/19/11…$15.00
https://doi.org/10.1145/3365609.3365866

1 INTRODUCTION

Networks have grown increasingly large and more complex as more services, users, and workloads move online. Consequently, the day-to-day operation of such networks, a difficult task to begin with, has become increasingly more challenging. The failure to properly manage and validate network state can lead to outages that impact large swaths of users, and indeed such failures happen frequently in practice [2, 8, 13–17].

These factors have contributed to a surge of interest in both the academic and industry communities in network *verification*, which promises to enable systematic validation of network invariants. In response, researchers have produced many innovative tools with high fidelity and performance, such as HSA [10], Veriflow [11], Batfish [6], ARC [7], and Minesweeper [3] to name a few.

However, the use of such tools is still not widespread outside of the few largest of organizations; and even in such organizations, we are aware of outages that have happened despite the use of network verification because network verification was not being used as effectively as it could have been. In particular, the network invariants were not testing in some critical aspects of the network.

We argue that the next frontier for network verification is, not more sophisticated analysis tools, but enabling easy and effective use of network verification by network engineers on the ground. We do not intend to imply that network verification is a solved problem from an analysis perspective; our position is that the primary bottleneck to adoption today is easy and effective use.

By way of analogy, in software development, many frameworks exist today that assist practitioners in writing robust code by making it easy to test their code for correctness and by measuring the effectiveness (i.e., coverage) of said tests [1, 9]. Coverage metrics, in particular, make it easy to spot gaps in existing tests, facilitating a feedback loop where users can add new tests to close these gaps.

While powerful network verification tools exist today, based on our experience with these tools and interacting with network engineers, we have found that they do not provide a comparable experience. In particular, the verification APIs offered by current tools are too low-level compared to the invariants that network engineers want to ensure. Further,

these tools cannot guide the user as to how "good" their tests are or what aspects of the network remain untested.

To address these hurdles, we present the design of a new network testing framework. A key challenge for this design is to develop a method to compute test coverage for networks. The coverage metrics should be able to inform users how well individual aspects of the network are tested and where they should focus their testing efforts. While such reports are inspired by the software domain, providing a meaningful notion of coverage presents a different challenge for networks. Unlike programs, where the flow of information is linear (i.e., statement X follows statement Y), networks consist of a bag on intertwined facts, e.g., the IP address of this interface determines if a BGP session will be established, which determines if a BGP route is learned, which determines if the route will appear in a forwarding table. We address this challenge by modeling the network as a dependency graph of "facts" and containers of facts. Our model is not specific to a particular tool but operates at the level of fundamental networking concepts (e.g., interface IP addresses, access control list entries).

Computing network coverage is also complicated by the difficulty of tracking what "facts" have been tested in an unrestricted setting. To address this issue, we propose a new domain-specific language for expressing network invariants. Our language simplifies the expression of invariants at a high level, by enabling users to read and filter information from different aspects of the network through a variety of operations over the network model mentioned above. However, more importantly, the declarative nature of the language makes it possible to accurately compute of coverage metrics.

Our language and coverage metrics can be integrated with existing verification tools. To illustrate such an integration, we report on the results of a small scale experiment in which we report coverage metrics for a network tested using high-level reachability invariants.

2 MOTIVATION

We motivate our position using a simple example that illustrates what it takes to verify reasonable properties in existing tools and what that experience lacks. Consider a standard three-tier datacenter, where the bottom tier is top-of-rack switches (ToR), the middle tier is aggregation switches, and the top tier is spine switches. Assume that all hosts within a rack belong to the same application (e.g., Web server or database server), and some applications may be spread across multiple racks. Also assume that each application has a well-known address space (e.g., 10.0.0.0/16 for Web servers).

Now suppose one of the invariants we want to ensure in this network is that all hosts of the Web server application can reach all hosts of the database application. From a network validation perspective, this invariant implies that if a packet with appropriate headers enter a ToR through an interface where a Web server is connected, it should be able to reach the right destination ToR which should in turn forward the packet out the interface to which the database server is connected. "Appropriate" headers could mean that the source IP is that of the source Web server, the destination IP is that of the destination database server, and the destination port is one of the database ports.

The invariant above can be checked with a reachability query that several verification tools support [3, 4, 6, 7, 10–12]. An abstract view of this query, without getting into specifics of individual tools, is that it checks for interface-to-interface reachability. Users can check if $reachable(s_i, d_i, h)$ is true, that is, if all possible packets that start at an interface in the set s_i and have headers field values in the space h can reach at least one of the interfaces in set d_i. This query will return false if any packet in the specified set is dropped by the network, for instance, due to a bad access-control list or routing protocol misconfiguration for destination addresses.

To use the query to check our invariant, we need to first identify all valid interfaces that connect to the source and destination application. Then, we need to find all headers that are specific to each source-destination interface pair. It is not correct to simply use source and destination addresses of the entire space that has been assigned to the applications; some addresses in that space may be unused (for which it is acceptable to not have reachability), and individual ToRs may be configured to provide connectivity to only the specific addresses that they connect to (and not the full space).

Pseudocode for this process is show in Figure 1. In the code, v.x() represents a call to the verification tool. In addition to the reachability query, we are using a query that returns all interfaces in the network. The first chunk of the code is identifying all relevant interfaces. The second chunk is running a number of reachability queries over pairs of interfaces using headers that are relevant to that pair and collecting all such counter examples. Each counterexample is a pair of interfaces such that a Web server host connected to the source interface will not be able to reach the DB server connected to the destination interface.

One problem with this approach is that the amount of effort needed to check even straightforward invariants is high. This effort is needed because of the semantic mismatch between the query, which takes constant values as inputs (specific interfaces and headers), and the invariant, which is expressed in terms of higher-level information that must be translated to those constants by the users. This state of affairs is particularly problematic because most network engineers are not expert software developers [5].

A more fundamental problem with this approach is that, given a test suite where each test is like the pseudocode

```
function testWebToDbReachability()
  webInterfaces = {}
  dbInterfaces = {}
  for iface in (v.networkInterfaces())
    if iface.ip in WEB_ADDRESS_SPACE
       webInterfaces.add(iface)
    if iface.ip in DB_ADDRESSS_SPACE
       dbInterfaces.add(iface)

  counterExamples = {}
  for webIface in webInterfaces
    for dbIface in dbInterfaces
       h = header()
       h.srcIps = webIface.subnet
       h.notSrcIps = webIface.ip
       h.dstIps = dbIface.subnet
       h.notDstIps = dbIface.ip
       h.dstPorts  = DB_PORTS
       if !v.reachable(webIface,dbIface,h)
          ce = [webIface, dbIface]
          counterExamples.add(ce)

  assert counterExamples.length() == 0
```

Figure 1: Example test using a verification tool API.

above, it is quite difficult to give feedback to the user as to how good is their test coverage. We formalize the notion of test coverage below but for now consider it to be a quantitative measure of how well different aspects of the network have been covered. Without a deep (and perhaps impossible) inspection of the user's test code, we can only tell that the user has a test in which they read IP addresses of all interfaces and then run a series of reachability queries. It cannot tell, for instance, that no checking was done at all for interfaces that did not match certain criteria.

The inability to provide guidance on test coverage is a key hurdle toward effective use of network verification. We know of cases where major outages have happened despite a robust use of network verification. These outages could have been prevented had the user been given some guidance of what their existing test suite did or did not cover.

3 TEST COVERAGE FOR NETWORKS

For software, code coverage metrics are frequently used to estimate how well a test suite is able to test a codebase. Coverage tools estimate these metrics by observing the test suite in action and measuring the fraction of the code base that is exercised by the user-provided tests. Multiple coverage metrics exist, which measure coverage for important concepts in the program (e.g., statements, basic blocks, branches, subroutines). The most basic one is to measure the fraction of statements that are covered. Test suites that do well on this metric cover a large fraction of program statements.

However, no such associated tools, or even metrics, exist for estimating network coverage given a test suite. In this section, we put forth a coverage metric that is inspired by statement coverage, and in the next section, we outline a system to estimate this metric. As with software, there are likely other valid metrics of test coverage, which we will explore as our work matures.

A key difference between the software world and networking is that the network configurations are driven by data (e.g., the loopback interface's IP address for a router is 17.0.121.3) rather than code. While programs execute sequentially (statement Y follows after statement X), networks resemble a bag of "facts" that are related in complex ways dictated by the various protocols that govern the behavior of the network.

One may wonder if it is possible to simply measure the coverage in the usual way, in terms of the code that runs on the router (e.g., Cisco iOS). However, there are several issues with this approach. The first is that proprietary router software source code is usually not available. The second is that a user will typically want to focus on the coverage of their network with respect to the configurations that they wrote. For instance, suppose a user never uses a particular feature of BGP in their code base, and therefore never writes tests for this feature. By analyzing the coverage of the router software, one would report code related to this feature as being not tested. And yet, the user in this case would be correct in not needing to write such tests.

Another important difference between the software world and networking is that frequently network tests are expressed, not in terms of information that directly appears in the configurations, but in terms of data that is derived from other data. For instance, the reachability query from the example is a test over the FIB (Forwarding Information Base) rules. Tools like Batfish [6] compute such entries from configurations. When FIB rules are tested, information that appears in the configuration are tested as well. A useful coverage metric should account for such dependencies.

3.1 Network model

Based on the observations above, for the purposes of computing coverage, we model the network as a dependency graph. Nodes in the graph correspond to either *facts* that can be tested and *containers* of one or more facts. One may think of facts as basic blocks in software and containers as entities that contain basic blocks such as functions, packages, and files; though the relationship between facts is different in the two domains.

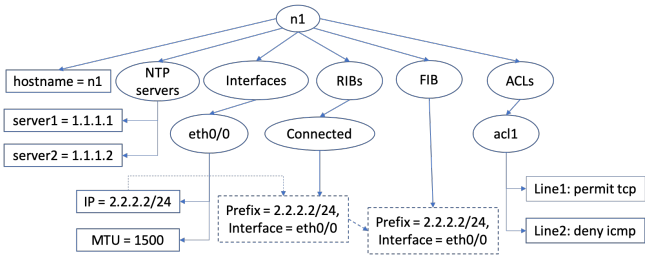


Figure 2: A simplified subset of a dependency graph.

Facts may appear directly in the input that is supplied to the verifier, or derived from the input. We call these two types of facts, respectively, as *base facts* and *derived facts*. Whether some network information is base or derived fact depends on the verification tool and its input. For instance, for data plane verification tools such as HSA [10], which take FIBs as input, FIB rules are base facts. For control plane verification tools such as Batfish, which take configuration as input and compute FIBs from it, FIB rules are derived facts. In the rest of the paper, unless explicitly stated otherwise, we assume that we are working with a control plane verification tool; those tools handle a superset of concepts compared to data plane verification tools.

Figure 2 shows a simplified subset of our dependency graph model for a control plane verifier. The ellipses denote containers, solid rectangles denote base facts, and dashed rectangles denote derived facts. Solid edges (which emanate only from containers) denote a containment relationship. Dashed edges (which lead to only derived facts) denote a data dependency; the target fact is derived from the source fact. There can be multiple dependency edges that lead to a derived fact.

In the figure, we see example facts corresponding to the node *n1*'s hostname, NTP servers, interfaces, RIBs (routing information base), FIB, and ACLs (access control lists). We also see a RIB entry fact for a connected route that is derived from the interface address, and a FIB entry that is derived from that RIB entry. In the full model, there will be many more RIB entries, including those for other protocols such as BGP, and many more FIB entries. The full model will also have dependencies that go across nodes, e.g., BGP RIB entries of *n1* may depend on BGP RIB and FIB entries of the neighboring node, in addition to depending on facts that capture relevant *n1*'s BGP neighbor configuration.

3.2 Coverage metric

Before we can define a test coverage metric atop our network model, we must formalize what a test execution does. This formalization then provides raw input to the coverage metric. In the software world, a test exercises one or more statements.

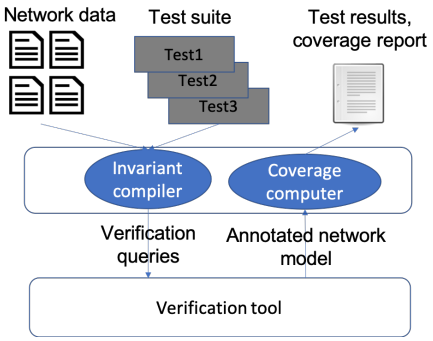


Figure 3: System architecture.

Similarly, in our context, we deem that a test exercises one or more facts, that is, it reads the value of those facts. For instance, a reachability query exercises facts for FIB entries and ACL lines on nodes along the path.

We can now define the coverage metric for a test as the fraction of all base facts that it covers. A base fact is considered "covered" if it is directly exercised by the test or if one of its descendants (derived) fact is exercised by the test. The coverage of a test suite is the fraction of all base facts collectively covered by the tests in the suite.

4 SYSTEM OVERVIEW

We now outline our network testing framework. Our system has two main components: 1) an invariant language and an associated compiler for declaratively specifying network tests, which enables (2) a dependency tracker and coverage estimator.

Figure 3 shows the system architecture. It takes as input network data and the test suite with individual tests written in our language, and it outputs test results along with the coverage metrics. The system is layered on top of existing verification tools, be they data plane verifiers such as HSA or control plane verifiers such as Batfish. To integrate with a tool, our compiler will translate the tests to one or more verification API calls that are native to the tool. This aspect of the integration requires no changes to the underlying tool. However, some instrumentation is needed to verification tools to track dependencies in the network model and label which facts in the model were covered by a test. The coverage computer takes this annotated network model to compute the output that is returned to the user. We discuss in §6 how to do dependency tracking in a scalable manner. But first we describe our language.

5 NETWORK INVARIANT LANGUAGE

Detecting what facts are covered by a test is challenging in a completely general setting. Recall that, in the reachability

example from Figure 1, the code inspects the address value for every interface in the network even though the behavior of on a small subset is actually tested. To simplify tracking coverage metrics and to make it easier to express network invariants, we now present a new, domain-specific testing language for our framework. The language lets users flexibly query, filter, and then check the state of facts (and containers) defined in the network model. We start by describing the language through a series of examples and then provide its formal specification.

5.1 Example: Counting NTP servers

As a simple first example, consider an invariant that all ToRs in the network should be configured with three NTP servers. This invariant can be expressed simply as:

```
nodes[hostname matches "tor.*"]
verify all count(ntp_servers) == 3
```

The test starts by looking through individual nodes contained in the "nodes" container where the node's hostname matches the regular expression "tor.*" (assuming that is the way to identify all ToRs in this network). The result of this filtering will be a new nodes container that contains a subset of the original nodes. The query then takes the resulting subgraph and verifies that all children (individual nodes) of the root (nodes container) have exactly three NTP servers.

5.2 Example: Web server/DB reachability

Our earlier invariant from §2, regarding the Web server to database server reachability, can be expressed as:

```
reachability
  [ingress anyof interfaces[ip in WEBSPACE]]
  [egress anyof interfaces[ip in DBSPACE]]
  [srcIp in ingress.subnet]
  [dstIp in egress.subnet]
  [srcIp != ingress.ip]
  [dstIp != egress.ip]
  [dstPort anyof DBPORTS]
verify all result == true
```

The query begins by looking through the (derived) reachability container that, conceptually[1], contains facts about the reachability of every possible ingress, egress, packet header pair. From reachability facts, it filters out those where the ingress is some web interface, the egress is some database interface, which are themselves provided by filtering the interfaces container for those interfaces that have ip addresses in these address spaces respectively. Further filtering restricts the source (destination) IP to be contained within the ingress (egress) interface subnet but is not the exact interface IP. Similarly, the destination port is filtered to be one of the database ports. The result in the dependency subgraph for the reachability container with this restricted set of packet headers ingress, and egress point identifiers. The test then verifies that the reachability result value is true for each remaining reachability fact.

[1]In verification tools, this container is an abstract view that is not actually materialized but rather reasoned about symbolically.

5.3 Example: BGP advertisements

As noted in §2, a challenge with verification tools today is that calls to the verifier must provide exact values (e.g., the prefix 125.3.17.0/24), making it hard write tests that relate different parts of the network. Another example of this is where the packets that users expect to be successful between two devices is likely related to other configuration objects such as the BGP networks configured to be advertised by the destination ToR in a data center.

Consider the invariant below that shows how our language can capture such an invariant:

```
reachability
  [ingress.name matches "T.*"]
  [egress.name matches "T.*"]
  [srcIp anyof ingress.local_ips]
  [dstIp in egress.bgp_networks]
  [protocol == UDP && dstPort == 53]
verify all result == true
```

The test starts by looking at the reachability container as with the previous example. The query filters the container to only include entries where a packet starts from a department router and ends at a border router. Further, it requires that the source IP address must be a "local IP" address defined for the ingress location; the destination IP address is covered by at least one prefix from the bgp_networks container at the egress location; and the packet protocol is UDP and destination port is 53 (DNS). Finally, we check that the result for all remaining reachability facts is true.

5.4 Language specification

Given the dependency graph model in §3, invariants are written using this simple language for filtering and extracting components of the graph. While traditionally verification tools have distinguished between symbolic (e.g., reachability) and concrete (e.g., BGP session) information, we represent and query both using the same abstraction.

The syntax for the language is shown in Figure 4. It defines a valid test case, which consists of an expression e followed by a collection of verification expressions that take a quantifier **all**, **some**, or **none** and an expression that should hold for either all, some, or no facts in the resulting subgraph. Expressions can filter a table where a number of other expressions hold (filter expression), refer to a constant or fully

t	::=	$e\ v_1, \ldots, v_n$	*test case*
v	::=	**verify** $q\ e$	*verify expression*
q	::=	**all** | **some** | **none**	*quantifier type*
e	::=	*constant*	*literal constant*
	|	fqn	*fully qualified name*
	|	$e_1 \circ e_2$	*binary op*
	|	$e_1[e_2, \ldots, e_n]$	*filter expression*
	|	**let** $x = e_1\ e_2$	*assignment*
	|	**count** e	*count expression*
	|	e_1 **matches** e_2	*regex match*
	|	e_1 **in** e_2	*ip subnet coverage*
	|	e_1 **anyof** e_2	*membership*

Figure 4: Invariant language.

qualified name (e.g., node.ntp_servers), compare two expressions for equality or inequality, assign an expression to a variable x (**let**), count the number of facts in a container (**count**), check if a string expression matches a regular expression given from a string (**matches**), combine expressions with a binary operation (∘), check if an IP address expression is covered by any prefix from a table of prefixes (**in**), and test if a value given by an expression is equal to any of a collection of values given by a table (**anyof**).

6 COMPUTING COVERAGE

To compute the coverage of a given test, we need the network model with facts and dependencies, along with information on which facts were directly covered for the test. From that information, we can derive which base facts were covered and, from that, the coverage metric.

We need to instrument the verification tool to get the information we need to compute coverage. What this entails depends on the tool. For data plane verification tools, which take FIB entries as input, this task is straightforward. Their network model has FIB entries as base facts and have no derived facts (or dependencies). From these tools, we just need to output which FIB entries were exercised by a test.

For control plane verification tools such as Batfish, which take configurations as input, the task of generating the information needed for coverage computation can be more involved. These tools compute forwarding state (FIB entries) from device configurations. The challenge is scalably tracking the dependency from configuration information to FIB entries. This computation is akin to provenance tracking, which is known to be a hard problem from both memory and compute perspectives. (The early version of Batfish, described in the original paper [6], was based on a Datalog engine that was tracking provenance, but that engine has since been replaced by an imperative engine because it did not scale to large networks.)

We can address this challenge by associating each base fact with a GUID (globally unique identifier) and each derived fact with a Bloom filter. Bloom filters are highly efficient data structure for tracking set membership such that we cannot enumerate the members in the set but can check if a given member is present in the set. The membership check can have a small false positive rate (i.e., members not inserted into the set may be inferred as being present) that depends on the size of the filter; the false negative rate is guaranteed to be zero. When a new derived fact is computed, its Bloom filter is a union of the set of GUIDs of base facts on which it is based and the Bloom filters of the derived facts on which it is based. Test execution yields information on which facts (base or derived) were covered. To go from that to which base facts were (indirectly) covered, we can check which base fact GUIDs are present in the Bloom filters of derived facts. This check will have a slight false positive rate but the overall computation will be a lot faster and lower overhead than precise provenance tracking.

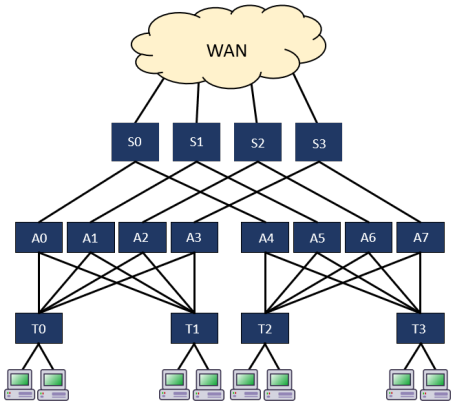
7 PRELIMINARY EVALUATION

To demonstrate the usefulness of network coverage feedback, we show coverage metrics computed for a simple datacenter for control plane verification (which takes network configuration as input and derives RIBs and FIBs based on it).

Consider the network in Figure 5(a). Assume that each ToR router will advertise two subnets into BGP, one for each of its connected hosts. Also assume that a default route is advertised from a wide-area network outside the data center to reach external destinations. After routing converges, each ToR will have 30 RIB entries–2 local BGP routes, the default route from each of the 4 connected aggregation routers, and 6 routes to other ToRs learned through 4 aggregation routers each. Each ToR will also have 36 FIB entries – 6 directly connected routes for interfaces, and 30 routes from the RIB (assuming multipath is configured).

Suppose our test suite consists of a single invariant, the one we described in §5.3–each ToR should be able to reach advertised BGP network by other ToRs. The query will evaluate the reachability of packets to the BGP subnets and we should thus consider these reachability facts as being covered. Since reachability is derived data, it will use provenance information to mark the 30 of the 36 FIB entries as covered–the two connected routes for hosts and the 4 default routes will not be covered. Since the FIB itself is derived, we will further cover 26 out of 30 of the RIB entries as being covered–all but the 4 default routes. This process would continue, and 4 of the 6 interfaces at T0 would be covered and so on. Other configuration elements on T0 such as NTP servers that are

(a) Network



(b) Coverage report

node	total elements	hit	missed	coverage
⊟ T0	80	60	20	75.0%
⊞ interfaces	6	4	2	66.6%
⊞ fibs	36	30	6	83.3%
⊞ ribs	30	26	4	86.6%
⊞ ntp servers	3	0	3	0.0%
...				
⊟ A0	60	54	6	90.0%
⊞ interfaces	3	3	1	100.0%
⊞ fibs	30	27	3	90.0%
⊞ ribs	27	24	3	88.8%
...				

Figure 5: Example coverage report (b) for a datacenter network (a) generated from evaluating the BGP network reachability test from §5 against the datacenter network.

not tested by the query will not be covered. Similarly, for aggregation routers, when checking reachability between ToRs, the FIB entries used to provide connectivity would be covered and a similar covering process would occur.

Figure 5(b) shows a subset of the coverage results from this test. It shows the total number of elements (facts), hits (covered facts), misses (uncovered facts), and coverage percent for each configuration element. Other nodes such as A0 are also summarized. One could easily drill down further into the data model to see finer grained information such as which interfaces have not been covered, or the coverage of different subelements in dependency graph such as the coverage of individual access-control lists.

Guidance for adding new tests. This coverage information makes it easy to understand both what is being tested currently as well as what new tests are needed in order to improve validation. We can clearly draw a few conclusions. First, the aggregation routers are being tested more thoroughly than the ToR routers, even though the reachability testing is from ToR-to-ToR. This is because ToR-to-ToR reachability relies on aggregation routers being correctly configured and ToRs have configuration related to connected servers that is not being tested. Second, the current test suite covers nothing that is not related to routing such as the NTP servers. This information will help a user realize that new tests that adding invariants that test for NTP servers and ToR-rack interfaces would be productive.

8 CONCLUSION

Network verification tools have made incredible progress towards scalable analysis of real networks. Yet effective use of such tools remains a challenge and hurts their adoption in practice. We argue that the next stage on the journey of network verification is to make such tools more accessible for average network operators. In other domains, such as software, where testing tools are widely used, code coverage is used as a useful metric for both measuring software test quality and to provide guidance on how to expand coverage with new tests. To build a similar experience for network operators, we formalize the definition of network coverage and present the design of a new invariant language. These elements simplify writing network tests and make it feasible to compute and report to users actionable network coverage statistics.

ACKNOWLEDGMENTS

The ideas presented in this paper have benefitted from conversations with several people. Yevgeniy Rombakh first mentioned the need for coverage metrics to guide the use of network verification; Dan Halperin and Victor Heorhiadi suggested ways to scalably track dependencies in Batfish; and discussions with Todd Millstein helped shaped the invariant language. We thank them all.

REFERENCES

[1] I. Ahmed, R. Gopinath, C. Brindescu, A. Groce, and C. Jensen. Can testedness be effectively measured? In *FSE*, 2016.
[2] M. Anderson. Time warner cable says outages largely resolved. http://www.seattletimes.com/business/time-warner-cable-says-outages-largely-resolved, 2014.
[3] R. Beckett, A. Gupta, R. Mahajan, and D. Walker. A general approach to network configuration verification. In *SIGCOMM*, 2017.
[4] S. K. Fayaz, T. Sharma, A. Fogel, R. Mahajan, T. Millstein, V. Sekar, and G. Varghese. Efficient network reachability analysis using a succinct control plane representation. In *OSDI*, 2016.

[5] G. Ferro. Turn network engineers into software engineers. https://etherealmind.com/turn-network-engineers-software-engineers/.

[6] A. Fogel, S. Fung, L. Pedrosa, M. Walraed-Sullivan, R. Govindan, R. Mahajan, and T. Millstein. A general approach to network configuration analysis. In *NSDI*, 2015.

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

[8] J. Godfrey. The summer of network misconfigurations. https://blog.algosec.com/2016/08/business-outages-caused-misconfigurations-headline-news-summer.html, 2016.

[9] M. Hilton, J. Bell, and D. Marinov. A large-scale study of test coverage evolution. In *ASE*, 2018.

[10] P. Kazemian, G. Varghese, and N. McKeown. Header space analysis: Static checking for networks. In *NSDI*, 2012.

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

[12] N. P. Lopes, N. Bjørner, P. Godefroid, K. Jayaraman, and G. Varghese. Checking beliefs in dynamic networks. In *NSDI*, 2015.

[13] S. Ragan. Bgp errors are to blame for monday's twitter outage, not ddos attacks. https://www.csoonline.com/article/3138934/security/bgp-errors-are-to-blame-for-monday-s-twitter-outage-not-ddos-attacks.html, 2016.

[14] S. Sharwood. Google cloud wobbles as workers patch wrong routers. http://www.theregister.co.uk/2016/03/01/$\text{google}_{\text{c}}\text{loud}_{\text{w}}\text{obbles}_{\text{a}}\text{s}_{\text{w}}\text{orkers}_{\text{p}}\text{atch}_{\text{w}}\text{rong}_{\text{r}}$outers/, 2016.

[15] Y. Sverdlik. Microsoft: misconfigured network device led to azure outage. http://www.datacenterdynamics.com/content-tracks/servers-storage/microsoft-misconfigured-network-device-led-to-azure-outage/68312.fullarticle, 2012.

[16] Y. Sverdlik. United says it outage resolved, dozen flights canceled monday. https://www.datacenterknowledge.com/archives/2017/01/23/united-says-it-outage-resolved-dozen-flights-canceled-monday, 2017.

[17] D. Tweney. 5-minute outage costs google $545,000 in revenue. https://venturebeat.com/2013/08/16/3-minute-outage-costs-google-545000-in-revenue/, August 2013.