Test Coverage Metrics for the Network

Xieyang Xu
University of Washington

Ryan Beckett
Microsoft

Karthick Jayaraman
Microsoft

Ratul Mahajan
University of Washington, Intentionet

David Walker
Princeton University

ABSTRACT

Testing and verification have emerged as key tools in the battle to improve the reliability of networks and the services they provide. However, the success of even the best technology of this sort is limited by how effectively it is applied, and in today's enormously complex industrial networks, it is surprisingly easy to overlook particular interfaces, routes, or flows when creating a test suite. Moreover, network engineers, unlike their software counterparts, have no help to battle this problem—there are no metrics or systems to compute the quality of their test suites or the extent to which their networks have been verified.

To address this gap, we develop a general framework to define and compute network coverage for stateless network data planes. It computes coverage for a range of network components (*e.g.*, interfaces, devices, paths) and supports many types of tests (*e.g.*, concrete versus symbolic; local versus end-to-end; tests that check network state versus those that analyze behavior). Our framework is based on the observation that any network dataplane component can be decomposed into forwarding rules and all types of tests ultimately exercise these rules using one or more packets.

We build a system called Yardstick based on this framework and deploy it in Microsoft Azure. Within the first month of its deployment inside one of the production networks, it uncovered several testing gaps and helped improve testing by covering 89% more forwarding rules and 17% more network interfaces.

CCS CONCEPTS

• **Networks** → **Network management**; **Network monitoring**; **Data center networks**; • **Computer systems organization** → **Reliability**; **Availability**; **Maintainability and maintenance**.

KEYWORDS

Coverage metrics, network verification, reliability

ACM Reference Format:
Xieyang Xu, Ryan Beckett, Karthick Jayaraman, Ratul Mahajan, and David Walker. 2021. Test Coverage Metrics for the Network. In *ACM SIGCOMM 2021 Conference (SIGCOMM '21), August 23–27, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3452296.3472941

SIGCOMM '21, August 23–27, 2021, Virtual Event, USA

ACM ISBN 978-1-4503-8383-7/21/08…$15.00
https://doi.org/10.1145/3452296.3472941

1 INTRODUCTION

Network outages and breaches have a huge cost for individuals and organizations alike, when essential services go offline, planes are grounded, and 911 emergency calls start failing [4, 27–29, 31]. In the last few years, network verification and other forms of systematic testing have emerged as keys to addressing this problem. These techniques have rapidly gone from research ideas to production deployments at all large cloud providers [6, 19, 30, 33, 36].

However, as we have learned from helping deploy verification for multiple networks, outages can happen despite heavy use of verification and testing when the users' test suites are incomplete and fail to test all important aspects of the network.

A glib suggestion to this problem is to say that engineers should develop better test suites, but that is easier said than done. Networks are complex and any long-running network has significant design heterogeneity accumulated over the years. Given a suite of tests, it can be nearly impossible for an engineer to judge how good the test suite is and what it does not test. Network engineers need better tools to make these judgements.

We draw inspiration from software testing to create such tools. The software domain has a range of coverage metrics that quantify the quality of test suites and provide insight into what aspects of the software are not well covered [2, 26]. Systems that compute and report coverage metrics are now an integral part of any software testing platform [8, 10, 17].

In networking, however, we do not even have well-defined coverage metrics, let alone practical systems to compute them. We must first define network coverage metrics. We cannot simply reuse software metrics given differences in the two domains. Software may be viewed as a graph of basic code blocks, where each block is a linear sequence of statements. A simple yet effective coverage metric is the fraction of statements or basic blocks tested. On the other hand, network forwarding state is a set of lookup tables atop a topology. The semantics of this state differs from linear statements. It also affords unique opportunities. Its inputs, for instance, are finite-length bit vectors (packets), which enables us to quantify the input space analyzed. Such quantification is much more difficult in general software systems with inputs of unbounded size.

A practical challenge that we face in defining and computing network coverage is that network testing comes in many flavors—it may directly inspect the forwarding state (*e.g.*, check that the default route exists [11, 33]), or it may validate that the forwarding state produces correct behavior (*e.g.*, devices forward a prefix in the right direction [19]); it may consider the behavior of a single concrete test packet individually (*e.g.*, via a traceroute [14, 35]), or it may consider the behavior of large sets of test packets (*e.g.*, via a symbolic simulation [21, 22]); and it may check individual devices [18], or it may check end-to-end paths [6]. At the same time, we need to compute a range of metrics that quantify how well different network components like devices, forwarding rules, interfaces, paths, and flows are tested. As with software, different metrics provide different lenses to analyze test suite quality, and they reveal different types of testing blind spots. For instance, if a test suite has high device-level coverage but low path-level coverage, it may not be testing an important device through which many paths traverse. Thus, we need a method to compute a range of metrics from a range of test types without a combinatorial explosion in the cost of testing or test analysis.

The coverage framework that we develop in this paper is based on the concept of an *atomic testable unit* of network forwarding state. An ATU is a pair of one packet and one forwarding state rule. It is the minimal unit that any test can exercise (though tests often exercise a rule using multiple packet or exercise multiple rules). The impact of individual tests and the whole test suite can be encoded using one or more ATUs. ATUs can also describe network components for which we want to compute coverage. The ATUs of a device include the cross-product of all of its rules and all possible packets, and the ATUs of a flow include all the rules it touches and its packet header space. Depending on which ATUs are covered by tests, a component may be fully tested, partially tested, or not tested at all. Thus, decomposing test coverage and components into ATUs provides a mathematical basis for computing a range of metrics from any type of test.

We build a system called Yardstick to compute coverage metrics based on this framework. It has an online phase during which testing tools report what they are testing using simple information that is readily available, and a post-processing phase during which it computes coverage metrics using this information. By splitting operation across these two phases, we ensure that metric computation, which can be expensive for large networks, is not on the critical network testing path—a practical concern for network operators.

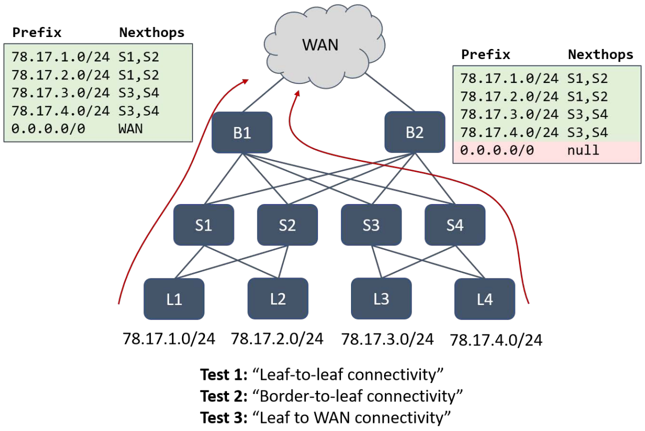
We deploy Yardstick in Microsoft Azure and integrate it with a testing tool that conducts all the types of tests mentioned above. Yardstick computes and reports several coverage metrics, providing feedback to network engineers on the quality of the test suites intended to evaluate if network changes are correct.

To demonstrate the value of Yardstick, we present a case study from the first month of its deployment in one of the production networks. Yardstick's coverage reports helped identify several testing gaps in this network. Certain types of rules and interfaces were not being tested. This information helped engineers develop two new types of tests that significantly improved coverage—by 89% for forwarding rules and 17% for interfaces. These new tests are now part of the production test suite.

We also benchmark the performance of Yardstick using synthetic data center networks of varying sizes. We find that the worst case increase in testing time is only 54 seconds, which is less than 3% for the test that takes 1967 seconds when it is not reporting coverage.

2 WHY COVERAGE METRICS

Consider the hypothetical data center network in Figure 1. It has a three-level hierarchy, with leaf routers at the bottom that connect to hosts (not shown), spine routers in the middle, and border routers at the top that connect to the wide-area network (WAN). The network runs the BGP routing protocol [23]. Each ToR has a prefix that it advertises inside the data center. The WAN announces the default route (0.0.0.0/0) to the border routers, which then propagate this route downward. The network is designed to withstand all possible single-node failures without disrupting application connectivity.



Prefix	Nexthops
78.17.1.0/24	S1,S2
78.17.2.0/24	S1,S2
78.17.3.0/24	S3,S4
78.17.4.0/24	S3,S4
0.0.0.0/0	WAN

Prefix	Nexthops
78.17.1.0/24	S1,S2
78.17.2.0/24	S1,S2
78.17.3.0/24	S3,S4
78.17.4.0/24	S3,S4
0.0.0.0/0	null

Test 1: "Leaf-to-leaf connectivity"
Test 2: "Border-to-leaf connectivity"
Test 3: "Leaf to WAN connectivity"

Figure 1: Test coverage for an example data center network. User tests check leaf-to-leaf, border-to-leaf, and leaf-to-WAN connectivity. Red arrows show the flow of packets from the leafs to the WAN. Forwarding table rules for B1 and B2 are shown, colored green if covered by a test and red otherwise.

Assume that the intended connectivity invariants are that bidirectional connectivity must exist between each pair of leafs and between each leaf and the WAN. Three tests are in place to check these invariants. The first checks that each leaf can reach each other leaf using packets with destinations addresses in the right prefix. The second test checks that each leaf can reach the WAN using packets with destination addresses outside the data center prefixes. The third test checks that each border router can reach each leaf using packets with the right destination addresses. With these tests, the network engineers believe they have all their bases covered.

However, they may discover that they do not when router $B1$ fails and the whole data center gets disconnected from the WAN, despite $B2$ being alive. It turns out $B2$ had a static default route that was null routed, which caused it to not propagate the default route to spines. So, when $B1$ fails, the spines have no route to the WAN.

How could the engineers in this scenario have uncovered the issue with their tests and prevented the outage? Once you know the exact root cause, many solutions suggest themselves. But instead of focusing on individual bug types, we need a general approach to uncover gaps in testing.

Coverage metrics can be the basis for that general approach. Suppose the engineers could compute *rule coverage* of their tests. Informally, this metric is the fraction of rules in the network through which least one test packet will pass; we formally define this metric later. Now, because no test packet uses the default route on $B2$, the coverage metric for $B2$ would have flagged the problem. It would have been lower than expected and also lower than $B1$, $B2$'s symmetrically-configured counterpart. The coverage metric could also have helped engineers improve the test suite. Once the underlying problem is discovered, the engineers could modify the test to not only check that the WAN is reachable from leaves but also that *all* spines and borders serve as conduits for this traffic.

3 METRIC COMPUTATION REQUIREMENTS

Network coverage metrics are intended to provide a sound basis for judging how well different network components such as devices, interfaces, and routes are exercised by the test suite. Exercising the component does not mean that all related bugs are caught. A test suite has two activities: *i*) exercising some components; and *ii*) asserting that resulting behaviors match expectations. To find bugs, the test suite must do both well. We focus on quantifying the quality of the first activity. This focus is similar to the software domain, where coverage metrics tend to quantify the fraction of statements or files exercised by the test suite and leave the task of judging the quality or assertions to other means.

3.1 Support diverse metrics and tests

We seek an approach to computing network coverage metrics that can support a diverse set of settings. Diversity refers not only to the types of networks (data center, backbone, etc.) and devices, but also to the types of metrics and tests. Let us elaborate.

Support for diverse metrics. For software, many different metrics have been devised to quantify coverage, such as the fraction of statements covered, the fraction of subroutines covered, the fraction of branches for which both paths are evaluated, and the fraction of control flow edges covered. Different metrics help software engineers focus on different aspects of coverage—for a given test suite, one metric may be high and another low, revealing a systematic testing deficiency. Different metrics may also represent different trade-offs in computation cost and bug-finding ability.

Networks too may be analyzed from many different perspectives. An engineer might ask: Do our tests cover every device? Every interface? Every path? Every flow? What do they say about a particular pod or about leaf routers? Each such question sheds light on a different type of testing gap. In the example above, we saw how rule coverage could identify the testing gap. This gap, however, would not have been revealed by device coverage, or the fraction of devices traversed by a test packet. Device coverage would have been 100% because every device was being traversed by at least one test. This includes *B*2, which was being covered by the border-to-leaf connectivity check (Test 2 in Figure 1).

Thus, our goal is not to devise a perfect coverage metric but support computation of a broad range of metrics. Doing so will enable network engineers to ask many different questions of interest and to drill down and investigate testing gaps.

Support for diverse tests. Network engineers use several types of tests. As shown in Figure 2, the tests can be broadly classified into *state inspection tests* or *behavioral tests*. State inspection tests directly inspect elements of the forwarding state and check that it matches expectations. An example is a test that checks whether the default route is present on a router. In contrast, behavioral tests analyzes the device or network behavior. An example is a test that executes a traceroute and then verifies packets emitted from a source can in fact reach a destination.

Behavioral tests can be further classified along two dimensions: *local* vs. *end-to-end* and *concrete* vs. *symbolic*. Local tests analyze individual devices, often by checking if a device forwards packets to a destination via certain interfaces. End-to-end tests reason about network behavior across a series of devices.

A concrete behavioral test such as a traceroute checks the behavior of a single, concrete packet. A symbolic test might check if *any* packet sent from a source will reach a destination—such tests reason about entire classes of packets. The terms "testing" and "verification" are sometimes used for these categories. In this paper, we use "test" to refer to all types of tests, and use "concrete" and "symbolic" to distinguish between the categories.

Multiple test types are often used for the same network because different types of tests have different strengths. State inspection tends to be faster; concrete tests tend to produce easier-to-understand results; symbolic tests tend to provide stronger guarantees; and local tests are more modular and efficient, while end-to-end tests provide better indication of whether high-level, network-wide invariants hold.

Coverage computation must thus support diverse types of tests. Test diversity poses a challenge, however. A basic function of a coverage metric is to enable users to judge if a new test will improve coverage compared to existing tests. This judgement is easy within the context of the same test type. For instance, compared to existing traceroute tests, it is easy to tell if a new traceroute test will exercise new network components, based on whether it uses a different source or packet headers. But it is hard to tell if a new traceroute test adds value compared to, say, existing symbolic tests.

3.2 Properties of metrics

To help users quantify and improve the quality of test suites, while supporting diverse networks, metrics, and tests, we devise metrics with the following properties.

Compositional. To compute multiple metrics for multiple test types in a tractable fashion, the metrics must be compositional in two ways. First, to seamlessly support diverse test types, equivalent sets of tests should yield equivalent coverage measures. Hence, we demand *i*) The coverage of a symbolic test must equal the combined coverage of a collection of concrete tests that collectively cover all those packets; *ii*) The coverage of a state-inspection test must equal the coverage of a symbolic test that considers all packets that can be impacted by that state. Second, we should be able to compute end-to-end coverage metrics (*e.g.*, for network paths) by composing the coverage metrics for a set of local tests, and compute local coverage metrics (*e.g.*, for individual devices) by decomposing what is tested by end-to-end tests.

We support such compositionality by mapping tests to a set of pairs of packet and forwarding state entry, a representation that is independent of the test type. We then compute all coverage metrics by taking a union of such sets. This uniform representation enables consistent treatment of different types of tests and prevents any double counting when multiple tests cover overlapping forwarding state entries.

State-inspection tests
Router R1's forwarding table must have the default route entry
Router R1's forwarding table must have an entry to prefix P with a next hop of neighbor
The access control list A1 on router R1 must have an entry that blocks packets to port 23

Behavioral tests		
	Local	**End-to-end**
Concrete	Router R1 must forward a given packet with dest. D via neighbor N1	Ping between two endpoints must succeed
	Router R1 must drop a given packet with dest. D and port 23	Traceroute between two endpoints must traverse the firewall
Symbolic	Router R1 must forward *all* packets to prefix P1 via neighbor N1	*All* packets in a defined set must succeed between two endpoints
	Router R1 must drop *all* packets to port 23	*All* packets between two endpoints must traverse a firewall

Figure 2: Taxonomy of network tests with examples of each type.

Semantics-based. Network coverage metrics should be based on the semantics of network state and independent of how devices process the state. Different devices may have different implementations for processing state. Take longest-prefix matching (LPM) as an example. One device may linearly scan the forwarding table (FIB) sorted by prefix length to find the matching entry, and another may use prefix-tries. Analogous to the software coverage metric of fraction of lines exercised by a test, we may consider the fraction of FIB entries inspected as our coverage metric. Per this metric, for a test packet that matches the default route (0.0.0.0/0) entry, the scanning-based device would have touched all the FIB entries and the trie-based device would have touched a handful of entries. Such device implementation-based metrics are undesirable as they are unlikely to be aligned with network engineers' expectations. In fact, the engineers may be completely unaware of internal device implementations. We thus develop metrics that are based on semantics of network state. For the case of a test packet that matches the default route entry, we should deem that only that entry is exercised.

Monotonic and bounded. Finally, the metrics should have certain basic numerical properties. In particular, they should be monotonic and bounded. A metric is *monotonic* if adding a test to an existing test suite never causes the metric to decrease. Formally, a metric *Cov* is monontonic when for all test suites T and individual tests t:

$$Cov[T] \leq Cov[T \cup \{t\}]$$

In other words, adding a new test never diminishes the value of a test suite (though it may not always strictly increase the value).

A metric is *bounded* if it varies between a well-defined minimum (0) and maximum (1). The minimum should correspond to the case where no tests are performed, and the maximum to the case where no further tests can possibly increase the value of the test suite. Boundedness helps the user gauge how far their current test suite is from full coverage, and it also helps compare coverage across different networks and across time for the same network.

Monotonicity and boundedness together imply that users can increase coverage by adding appropriate tests to a test suite whose coverage is currently less than ideal. And if test suite coverage does increase, it covers more of the network state, thus increasing the probability of uncovering more bugs in the system.

4 DEFINING NETWORK COVERAGE

Our network coverage metrics are based on general models of the network forwarding state and tests. We first describe these models and then our coverage computation framework. Figure 3 summarizes our notation.

4.1 Network Model

A network N is a 4-tuple (V, I, E, S). V is the set of network *devices*, and I is the set of *interfaces* on those devices. A network *location* ℓ is a pair, written $v.i$, of a device v and interface i. The set E contains links that connect locations. Finally, S, represents network forwarding state. For simplicity, each device contains a set of *rules* (R). We write $S[v]$ for the set of rules associated with a device v. A more sophisticated device model will have multiple tables of rules (*e.g.*, forwarding rules and access-control rules). Such extensions are straightforward but not necessary for explaining the coverage concepts that are our focus.

Rules operate over *located packets* (p), which include their location (ℓ) as well as the contents of their header fields. We use P to denote a set of located packets. A rule r will match some set of packets, called its *match set*, and apply an *action* to modify the matched packets. We write $M[r]$ for the match set of the rule r and $F[r]$ for the action of r. The match sets of rules do not overlap, making the rule that applies to any packet unambiguous. In practice, rules are ordered within tables and their match fields may match overlapping packet sets. The first rule in a table to match a packet is applied to the packet. In our model, such rules have been preprocessed to eliminate overlapping match sets; our implementation computes these match sets.

Possible actions of a rule include forwarding the packet via one or more interfaces, dropping the packet, and transforming some subset of header fields. In general, $F[r][p]$ is a set of located packets P. If P is empty, the input packet p has been dropped. If P contains a single located packet p', the input packet has been forwarded and possibly transformed. If P contains multiple packets on different interfaces, the device has multicast the original packet. To apply an action to a set of packets P, we apply the action to all packets $p \in P$ and take the union of the results.

Model limitations. Our model has static view of the network forwarding state and assumes that all tests were run on this state. This view is consistent with how all data plane verification tools operate—they take a snapshot of network state and run all queries

Notation	Description
r/R	A match-action rule/rule set.
$S[v]$	The set of rules associated with device v.
p/P	A located packet/packet set.
t/T	Network test/test suite.
$M[r]$	The match set of the rule r.
$F[r][p]$	The resulting packets after applying rule r on packet p, where $F[r]$ is the action of the rule r.
$T[r]$	The set of packets used to exercise r by test suite T.
$P \triangleright r_1, \ldots, r_j$	Guarded string, describing a flow of packet set P along the path $r_1, \ldots, r_j$, see §4.3.1.
(P_T, R_T)	Coverage trace of test suite T, a tuple of a packet set P_T and a rule set R_T, see §5.2.

Figure 3: Network coverage concepts and notations.

on it. But it may lose precision for live network tests (*e.g.*, ping or traceroute) if the state changes during test suite runs.

4.2 Modeling Network Tests

A test, be it a concrete traceroute or a symbolic analysis, checks whether the network handles some packets correctly by 1) consulting the network forwarding state, 2) computing packet transformation and forwarding, and 3) comparing the result with user-specified expectations. Coverage is determined by analyzing the forwarding state consulted in the first step and the set of packets considered in the second step. An Atomic Testable Unit, or ATU, (r, p) is our primitive measure of coverage—it indicates a test has exercised rule r on located packet p.

We model a test t as a (total) function from rules to sets of packets. When $t[r] = P$, we say that the test t has exercised rule r using packets P. Alternatively, we say that test t *covers* packets P for rule r. When $P = \{p_1, \ldots, p_k\}$, the corresponding set of ATUs is $\{(r, p_1), \ldots, (r, p_k)\}$. Note that the set of packets P cannot exceed the match-set of r.

The *covered set of a test* is the union of all packets covered for any rule r. That is, the covered set for t is $t[r_1] \cup \cdots \cup t[r_k]$ when $r_1, \ldots, r_k$ is the set of all rules in the network. Likewise, the ATUs for t are $\{(r, p) \mid v \in V, r \in S[v], p \in t[r]\}$.

To illustrate these concepts, when $t[r]$ is the empty set, r has not been exercised by the test at all. When $t[r]$ equals the match set of r, r has been completely tested. Often, $t[r]$ will be somewhere in between, indicating that a rule has been partially tested. For example, when representing a traceroute test, $t[r]$ will be $\{p\}$ for some packet p for each rule r along the traceroute path and empty set for each rule r not on the path. When representing a symbolic test, $t[r]$ may consist of many packets rather than just one.

A test suite T is simply a set of tests: $\{t_1, \ldots, t_k\}$. In a slight abuse of notation, we often treat test suites as functions from rules to packets. Applying a test suite to a rule yields the union of the packets tested by all tests in the suite:

$$T[r] = t_1[r] \cup \cdots \cup t_k[r]$$

The ATUs for a test suite are defined in the obvious way as the union of the ATUs of the underlying tests in the suite.

Model discussion. An ATU represents the finest granularity of the impact of a test. An alternative is to use rules as atomic units. However, that would have rendered the coverage framework unable to distinguish between a concrete test, like a traceroute, that exercises a portion of the rule with a single packet and a symbolic test that exercises more of the rule over many packets.

A limitation of our model is that it cannot account for coverage of stateful networks accurately. If a rule r uses a switch register or another stateful component, exercising it once on a particular packet may not suffice to test it completely. Since ATUs (r, p) do not track the state space covered by applications of r to packets p, our coverage metrics will be blind to the amount of the state space covered. We followed common software-testing frameworks in making this choice. They too typically measure coverage in terms of lines of code rather than state spaces covered per line of code. In both networks and software, tracking the state space covered may be prohibitively expensive. Thankfully, many network data planes are stateless, obviating the need to track state in such contexts.

4.3 Coverage Metrics

Given the models of the network state and tests, we can now define coverage metrics. Given our goal of supporting a diverse range of metrics, we developed a common framework for computing coverage for a variety of network components such as rules, devices, and paths. Below, we first describe this framework and then how different components map to it.

4.3.1 *A framework for computing coverage.*

Our framework computes the coverage of an individual component (*e.g.*, a device or rule); the coverage of multiple components of interest can then be aggregated (§ 4.3.3). The specification to compute the coverage of a network component has three parts:

- a *dependency specification* G,
- a *coverage measure* μ, and
- a *combinator* κ.

The dependency specification G describes the dependencies of a component—what needs to be tested to test that component. Specifically, $G = \{g_1, \ldots, g_k\}$ is a set of *guarded strings*, where a guarded string $g = P \triangleright r_1, \ldots, r_j$ is a located packet set P followed by a list of rules $r_1, \ldots, r_j$.[1] The packet set P is the guard. The network engineer is interested in the flow of those packets along

[1] Guarded strings are a natural unit of interest in network data planes. They were also used to provide semantics to NetKAT expressions [3].

the path $r_1, \ldots, r_j$ where r_1 through r_j are rules that form a valid path (i.e., r_i forwards to r_{i+1}).

If the engineer is interested in understanding coverage of a single device with three rules then the dependency specification might be $\{P_1 \rhd r_1, P_2 \rhd r_2, P_3 \rhd r_3\}$. In this case, each "path" is only one rule as long as we are not interested in multi-rule paths. The packet spaces P_1, P_2 and P_3 might be the match sets of the rules.

The coverage measure μ evaluates the extent to which a test suite T covers a guarded string g in G. It must be a function from T and g to a number between 0 and 1, with higher values implying higher coverage. For instance, given $g = P \rhd r$, a measure might return 1 if there exists a test that exercises the rule r on a packet $p \in P$ and 0 otherwise. Alternatively, a measure might return the ratio of packets $p \in P$ that a test exercises on r. When considering a path $g = P \rhd r_1, \ldots, r_j$, a measure could determine the minimal fraction of any rule's match set in the path covered by a test.

Finally, the combinator κ produces an overall coverage metric for a component by mapping sets of measures to measures. Such combinators might compute the average of the given measures or use the min or the max value to report coverage for the component; as discussed below, different choices are appropriate for different sorts of component.

Formally, given combinator κ, measure μ, and specification G for a component, the coverage of test suite T for that component is:

$$CompCov[T](\kappa, \mu, G) = \kappa\,(\text{map}\,(\mu[T])\,G) \tag{1}$$

The function $\text{map}(f)(S)$ applies f to every element of the set S.

Since network operators are interested not only in coverage of a single device or flow, our framework also defines coverage over collections of components (e.g., all devices in the network). This coverage of is programmable as well. Given an *aggregator* α and a specification of the collection $C = \{(\kappa_1, \mu_1, G_1), \ldots, (\kappa_k, \mu_k, G_k)\}$, we can define the overall coverage of a test suite as follows.

$$Cov[T](\alpha, C) = \alpha\,(\text{map}\,(CompCov[T])\,C) \tag{2}$$

§4.3.2 illustrates the use of Equation (1) by providing concrete specification for common network components, and §4.3.3 describes some useful aggregation functions for Equation (2) to summarize coverage for multiple components.

4.3.2 Coverage for common network components.

We show how to analyze coverage for the most common network components: rules, devices, interfaces, paths, and flows. Other components, such as the CoFlows [9] of a distributed application (*i.e.* the set of flows generated by the application) or all traffic traversing a firewall, can be analyzed similarly.

Rule coverage. Rule coverage quantifies the extent to which a network rule is covered by a test suite. Given a rule r, the dependency specification is $G = \{M[r] \rhd r\}$. The coverage measure adopted is $\mu = \frac{|T[r]|}{|M[r]|}$, which quantifies the fraction of the rule's match set covered by the test suite. This ratio will always be less than one because $T[r] \subseteq M[r]$. The combinator κ for this metric simply picks the only element in this singleton set.

Device coverage. Device coverage quantifies how well the forwarding state of the device is covered. Given a device with k rules, its dependency specification is $G = \{M[r_1] \rhd r_1, \cdots M[r_k] \rhd r_k\}$. Device coverage uses the same measure μ as rule coverage. Its combinator κ is the weighted average, where the weight is proportional to a rule's match set, that is, the weight for r_i is $\frac{|M[r_i]|}{\Sigma_{1 \le j \le k} |M[r_j]|}$. This way, device coverage reports the fraction of total packets against which the device as a whole has been tested.

Interface coverage. Interface coverage quantifies how well the state responsible for packets leaving or entering an interface is tested. For instance, engineers may evaluate outgoing interface coverage of border interfaces when they are interested in packets leaving the data center. The coverage specification for an interface is similar to that of a device except that the set of rules considered is limited to those relevant to the interface. For an outgoing interface, this set has all the rules that forward packets to the interface. For an incoming interface, it has all the rules that have the interface in their match sets, as we limit the corresponding packet guards to only those on the interface.

Path coverage. A path is a valid sequence of rules and path coverage is intended to quantify how well the forwarding state along a path is tested. The dependency specification of the path is $G = \{P \rhd r_1, ..., r_k\}$, where $r_1, ..., r_k$ is the sequence of rules that define the path, and P is the full set of located packets that can traverse the path. This set is not known apriori but can be compute by processing the forwarding state. It includes packets whose headers may get transformed along the way. It does not include packets that are dropped at an intermediate rule $r_{j<k}$ in the sequence; those packets will be part of the $r_1, ..., r_j$ path.

The measure μ of a path quantifies the fraction of P that has been tested through the entire sequence of rules. If different rules of the path were tested using disjoint sets of packets, the coverage will be zero, as no one packet has made it all the way across the path. This calculation proceeds rule by rule and computes at each hop the set of packets that remain under consideration. Formally:

$$P_i = \begin{cases} M[r_1] & \text{if } i = 0 \\ F[r_i][P_{i-1} \cap T[r_i]] & \text{otherwise} \end{cases} \tag{3}$$

P_i is the set of packets after rule r_i has been processed. In the beginning, it is $M[r_1]$ (match set of r_1). After processing r_i, it changes to the set obtained by applying the action of r_i to the intersection of P_{i-1} and r_i's tested set. The final coverage is $\frac{|P_k|}{|P|}$, the fraction of packets left at the end.[2]

Paths have only one guarded string, and hence κ selects the only element of this singleton set as its result.

Flow coverage. A flow can be defined using its source location and header space. When this flow is injected into the network, it traverses one or more paths (due to possible multi-path routing or different headers in the header space being forwarded differently). The dependency specification of the flow has a guarded string corresponding to each of its paths, and each such string has the flow's packets as the guard. The coverage measure is the same

[2] We have presented a simplified view that assumes that any packet transformations are one-to-one, so rule application preserves the number of packets in the set. To support one-to-many and many-to-one transformations, our system does a slightly more complex computation. It computes another sequence of packet sets P'_i that is not constrained by $T[r_i]$, by replacing it with $M[r_i]$ in the equation. It then computes $\frac{P_i}{P'_i}$ at each hop and takes the minimum ratio across all hops.

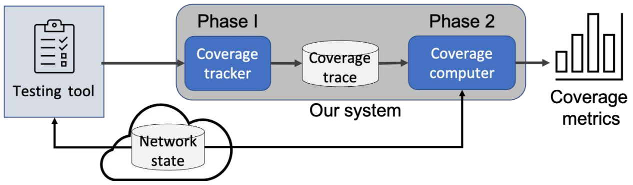


Figure 4: Overview of Yardstick.

as that for a path. The combinator is weighted average, where the weight is proportional to fraction of flow's packets that will use a path, computed by processing the network state. We are thus computing the fraction of flow's dependencies that have been tested end-to-end. If the coverage is 75%, it means that state corresponding to 75% of the flow's packet stream has been tested end-to-end.

4.3.3 *Coverage for component collections.*

We illustrated above how to compute coverage for individual components, but users will commonly be interested in aggregate coverage across multiple components of the same type. We currently support three forms of aggregator α, each of which provides a different perspective on how well components in the collection are covered.

- *Simple average:* Compute the (unweighted) mean coverage across components. Average device coverage will thus be the mean of the coverage of all devices under consideration. This aggregation provides a simple, overall view of how well devices are covered.
- *Weighted average:* Compute the weighted average coverage across all components, where the weight is the size of packet space handled by the component. This aggregation gives a higher weight to components that handle more packets. Rules that match more packets (e.g., the default route) will thus have a higher weight, encouraging engineers to cover them with a higher priority.
- *Fractional coverage:* Often we simply want to know what fraction of components have been tested *at all.* The fractional coverage aggregator maps the coverage of individual components in the collection to 0 or 1, based on whether the coverage is 0 or non-zero, and then computes the mean. If fractional device coverage is 80%, then 20% of the devices are completely untested and users would likely want to investigate why.

5 YARDSTICK DESIGN

Our system, Yardstick, is based on the formulation above. Its design is guided by the need to minimize impact on testing performance. This goal is important because some testing tools are in the critical path of network updates [20, 22] and testing delays will delay updates. In other settings, testing performance dictates how quickly a network state bug will be caught after it appears.

With this consideration, we split the operation of Yardstick across two phases, shown in Figure 4. The first phase tracks coverage reported by the testing tool, using two simple API calls:

- `markPacket(P)`
- `markRule(r)`

The `markPacket` call is used for behavioral tests, to report the located packets P used in tests. The `markRule` call is used for state inspection tests, to report which rules in the network are inspected.

The coverage tracker stores the information provided by testing tools in a compact representation, called the *coverage trace* (§5.2). The format of the coverage trace enables memory and time efficiency for both tracking and computing phases.

After testing finishes, in the second phase, Yardstick uses the coverage trace and network state to compute the requested coverage metrics. Since all the data is available, the network engineer can at any time ask the system to compute new metrics and zoom in from aggregate to individual component metrics.

Yardstick can be used with any testing tool that can be instrumented to call its APIs. Originally, we attempted a design that did not use explicit APIs such as `markPacket`, but snooped state read operations and registered coverage for elements of the state read during testing. This design assumed that all state elements read by the tool are exercised. Unfortunately, however, symbolic testing often requires that state be read in bulk up front to build internal data structures. It may not necessarily exercise all the data read.

Our chosen APIs enable easy integration with a range of testing tools. The information they need is readily available. Yardstick takes on the work to translate this information into what is needed for coverage computation. We discuss this next.

5.1 Using Coverage Tracking APIs

Different types of tests described in §3.1 can use our APIs to report coverage information with minimal overhead.

State inspection tests. These tests inspect forwarding state rules, *e.g.*, check whether a default route exists on a device. The coverage of inspecting a rule is its match set, so the information we need for metric computation is $t[r] = M[r]$.

However, $M[r]$ is not readily available to the testing tool (because the match field of the rule is not necessarily its match set, given earlier overlapping rules in the table) and it can take time to compute for large tables. Yardstick thus requires tools to just report the tested rule via `markRule(r)`, and it then computes $M[r]$ in the second phase to minimize the burden on the testing tool.

Local behavioral tests. These tests inject a located packet set P into the device and then validate that the device processes (drops or forwards) the packets as expected. Different testing tools compute the result of the device processing differently; it could be concrete simulation, symbolic simulation, or SMT constraints. Regardless of the method, the information we need from the perspective of coverage, is which rules were exercised using which subsets of P (given P may exercised multiple rules).

Unfortunately, this information is not computed by all tools as part of their testing, and computing it can be expensive. Yardstick thus requires the tools to just report P using `markPacket(P)` and derive the rule-level coverage information from it later.

End-to-end behavioral tests. Coverage for these tests is reported using `markPacket(P)` as well but a separate call is made for each hop in the network with the packet set at that hop. We did consider an alternative in which testing tool would report the packet set at only the origin instead of each hop and we would then infer the hop-level information. But we deemed that alternative not worthwhile because the tools already have hop-level information available.

Operation	Description
`PacketSet empty()`	Return an empty set of packets
`PacketSet negate(P)`	Return the set of packets not in the input set
`PacketSet union(P1, P2)`	Return the union of two packet sets
`PacketSet intersect(P1, P2)`	Return the intersection of two packet sets
`Boolean equal(P1, P2)`	Return if two packet sets are equal
`PacketSet fromRule(rule)`	Convert the match field of a rule to the corresponding set of packets
`Long count(P)`	Return the number of packets in the set

Figure 5: Operations over packet sets to help compute coverage.

Algorithm 1: Compute covered sets $T[r]$

```
Input: Network $N = (V, I, E, S)$, Test suite $T$
Procedure ComputeCoveredSets()
    for r $\in \bigcup_{v \in V} S[v]$ do
        if $r$ in $R_T$ then
            $T$[r] $\leftarrow M$[r]
        else
            $T$[r] $\leftarrow$ intersect($P_T, M$[r])
    return $\{T[r_i] \mid r_i \in \bigcup_{v \in V} S[v]\}$
```

5.2 Computing Coverage Metrics

During test execution (Phase 1), Yardstick stores the union of all information reported by the testing tool in the coverage trace. The trace can be represented using the tuple (P_T, R_T), where P_T is the set of located packets across all `markPacket` API calls, and R_T is a set of rules across all `markRule` API calls. Yardstick does not keep the entire log and removes overlapping information on the fly.

After test execution (Phase 2), Yardstick uses the coverage trace to compute the requested metrics. This computation can be described using operations over packet sets shown in Figure 5. We compute coverage in three steps.

Step 1: Compute rule match sets. We first compute match sets of rules using the forwarding state. Our network and test models have disjoint match sets for rules in the same table. Given a device v and its forwarding rules $S[v]$, we compute the disjoint match set of each rule by walking the ordered list of device rules, computing the match set of each as the intersection of its match field and packets not matched so far [32].

Step 2: Compute covered sets. From the coverage trace (P_T, R_T), we compute the covered sets of all rules in the network using Algorithm 1. If a rule exists in R_T (reported by inspection tests), its covered set is its match set. Otherwise, it is the intersection of its match set and tested packets P_T.

Step 3: Compute coverage metrics. In the final step, Yardstick computes the component-level and aggregate coverage metrics, using the framework of §4.3. These computations are straightforward for all metrics except for path-based metrics. Unlike other metrics, information about neither the set of covered paths nor the number of all paths is readily available in the coverage trace. For boundedness (§3), we need the number of all paths as the denominator for aggregate metrics–if we see 10 paths in the coverage data, is that out of 10 possible paths (100% covered) or 1000 (1% covered)? This count may be known for some types of structured networks but not in the general case.

When the count of paths is not known apriori, we consider all possible paths imputed by the network forwarding state as the total number of paths. All possible paths cannot be computed based on topology alone because many unrealistic zigzag paths (*e.g.*, a 20-hop leaf-spine-leaf-spine$\cdots$ path) will inflate the path count. We thus use the network forwarding state to compute paths that carry non-zero traffic. Computing the path count in this manner has risks because network state bugs can change the count.[3] We can guard against this risk by flagging to the user when the size of path universe changes dramatically relative to prior state snapshots. Absent major operational changes to the network, this universe is not expected to change significantly from day-to-day.

We compute the path universe by symbolically exploring the journey of all possible headers from all starting locations. This traversal is depth-first on the topology and emits new paths incrementally (i.e., first a single-hop path with the source, then a two-hop path with the source and its first neighbor, assuming that the source sends a non-empty set of packets to this neighbor, and so on). We do not store all paths in memory–there can be 100s of millions of paths in a large network—but process them on the fly.

Processing a path involves computing its coverage using the coverage trace per Equation (3). When a path is emitted, we know its sequence of rules. We do not know its guard set of packets. Strictly speaking, we do not need to know which packets are in the guard set; we only need to know the size of this set. In the common case, when any transformations along the path are only one-to-one, this size is the same as that of the final packet set at the end of our path exploration, and we use that size. In other cases, we compute the guard set by reversing the forwarding operations using the final set of packets. At each hop this computes the input set of packets that can produce the output set. We encode rule actions as BDDs, which allows for quickly doing such computations.

6 SYSTEM IMPLEMENTATION

Yardstick is implemented using 2300 lines of C# code, not counting the lines in various third-party libraries. The packet set operations in Figure 5 are implemented using binary decision diagrams (BDDs) that can efficiently encode and manipulate large header spaces.

[3] Path-based metrics in the software domain have the same risk.

We also instrumented a testing tool to report coverage to Yardstick. This tool is deployed in Microsoft Azure and supports all the test types mentioned in §3.1. Because the information Yardstick needs is readily available, we had to change only 7 lines of tool's code to report coverage. Each such line was an API call inserted at an appropriate place in the testing logic. Yardstick links to the tool as a dynamic library, and the two share type definitions for objects such as packet sets. Such an integration helps eliminate serialization-deserialization overhead. (Other tools integrated with Yardstick would likely incur higher data processing overhead.)

By default, Yardstick computes coverage for all device, interface, and rule coverage in the network. Users can customize the coverage report in a few different ways. They can request path coverage, which is not computed by default because of its high computational cost (§8). Users can also narrow the coverage analysis to particular flows by specifying the flows' start locations and header spaces.

Finally, users can zoom in on a subset of components by providing a binary function that takes in the component and returns true if that component should be considered. This filtering option is helpful when users want to analyze coverage for, say, only leaf routers or only inside-to-outside paths. In the future, we plan to let users provide a coverage specification directly, allowing them to compute coverage for a richer set of component types.

7 CASE STUDY

Yardstick is deployed in Azure as part of a service to evaluate the impact of changes to production networks. The service computes the network forwarding state that results from the change and then uses a test suite to check if the state is correct. We present a case study based on one month of Yardstick deployment in one of the networks. Its coverage reports identified systematic testing gaps in the original test suite and helped improve the test suite by suggesting new tests to address those gaps. These new tests are now part of the network's test suite.

7.1 Network Overview

Our case study focuses on a regional network that interconnects hundreds of thousands of hosts across multiple data centers in the same geographic region.

Topology and routing. Each datacenter network is a hierarchical Clos topology [13]. The top-of-rack (ToR) routers are at the bottom, and they directly connect to hosts. Aggregation routers connect the top-of-rack routers together in pods, which in turn are connected via a set of spine routers. At the top of each data center, the spine routers are connected to multiple layers of regional hub routers [24] which interconnect datacenters within a region. The regional hub routers are further connected to wide-area backbone routers that provide connectivity to the Internet and to other regions.

The network uses eBGP routing protocol on all routers [23]. Each router is assigned a private BGP ASN based on its role in the datacenter and configured with the `allow-as-in` command to avoid rejecting valid paths as loops [23] (*e.g.*, a ToR1-Aggregation1-ToR2 path is legal even though the two ToRs have the same ASN). As a fail safe, every router is also configured with a default static route (for the prefix `0.0.0.0/0`) that forwards packets to connected, higher-layer ("northern") neighbors. In the event of many kinds of failures, this backup measure ensures that packets will still have connectivity. For redundancy and load balancing, eBGP equal cost multipath (ECMP) is enabled on all routers.

Each ToR connects to hosts via Ethernet interfaces with assigned subnets, and, to enable connectivity to those hosts, it advertises aggregated prefixes for its directly connected hosts into the eBGP routing protocol. In addition, each router has one or more loopback interfaces whose corresponding connected routes are injected into eBGP via route redistribution.

Testing Pipeline. The network undergoes frequent changes in response to planned maintenance, migrations, and policy updates. All major changes are rigorously tested prior to deployment using a two-step process. An in-house simulator [25] and emulator [24] compute the network forwarding state that will result from the change. This state is then tested using a test suite defined by network engineers. Test execution produces a pass-fail report that network operators analyze to determine if the change is safe. Human oversight is needed here because it is possible that tests may fail as a result of modeling error or transient failures.

Yardstick has been integrated into this pipeline, where it augments the test results with coverage metrics. Its output is analyzed by a network engineer to improve the test suite and by network operators to determine the safety of the change.

7.2 Identifying Testing Gaps

Prior to the introduction of Yardstick, the network's testing pipeline used a collection of tests of two types:

1. *DefaultRouteCheck:* a subset of RCDC [19] contracts related to the default route that check that default routes have the correct set of next hops. This a state-inspection test.
2. *AggCanReachTorLoopback:* check that the aggregation routers correctly forward packets for ToR routers' loopback interfaces. This is a local symbolic test.

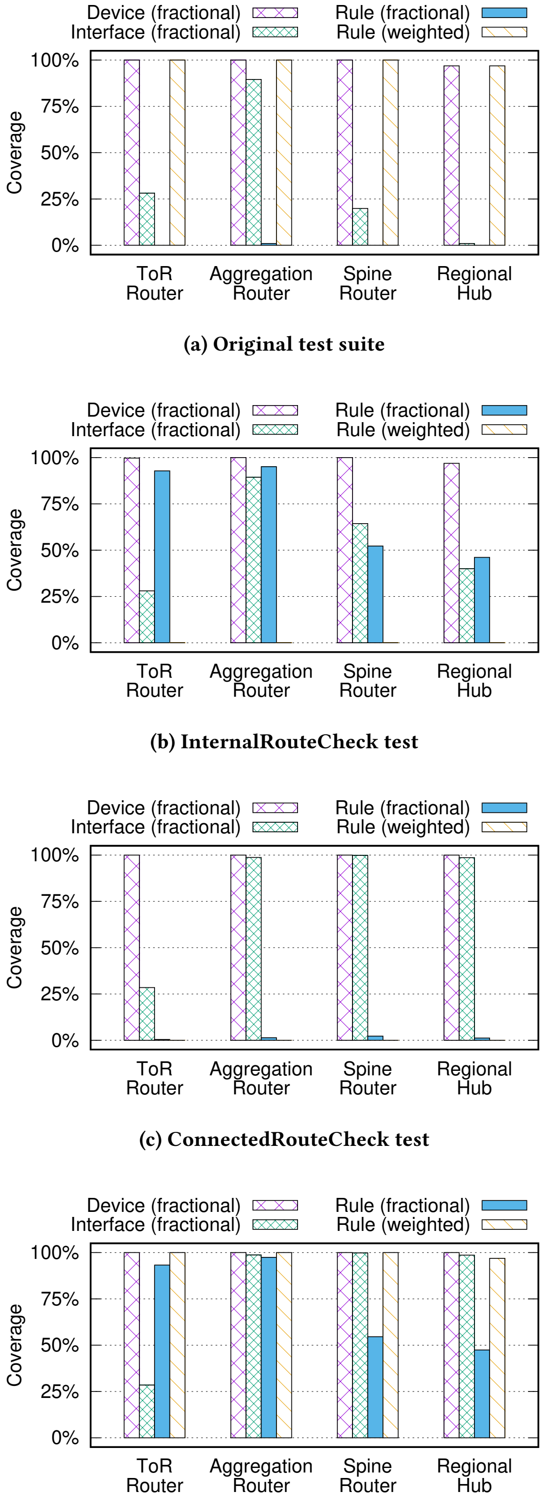
Figure 6a shows the coverage that Yardstick reported for this test suite. We break results by router type and plot fractional averages (§4.3.3) for devices, rules, and interfaces. We also plot the weighted average for rules; weighted average for devices and interfaces (not shown) was similar to that for rules. This view of the data was particularly useful toward understanding testing effectiveness and gaps for this network.

We see that fractional device coverage is close to perfect for all types of routers.[4] This high coverage is because the DefaultRouteCheck covers all devices. Device coverage is slightly low for regional hub routers because some of those routers are not expected to have the default route, and so the test excludes them.

Interface coverage, on the other hand, is quite uneven. It is high for aggregation routers because of the AggCanReachTorLoopback test but low for other router types. If the default route is not using the interface, it is not being tested at all. Thus, primarily northbound interfaces (*i.e.*, toward the higher layers in the hierarchy) are tested.

Rule coverage tells an interesting story and also shows the value of different types of aggregations. Fractional rule coverage is really

[4] Yardstick did actually reveal some completely untested devices in the network. This discovery was initially surprising to engineers. It later became clear that the untested devices were legacy routers that could not be tested. We have excluded such devices from our analysis.



(a) Original test suite

(b) InternalRouteCheck test

(c) ConnectedRouteCheck test

(d) Final test suite

Figure 6: Coverage for different tests.

low—most of the bars for this measure are indistinguishable from 0 in Figure 6a—indicating that the vast majority of the rules are untested. But weighted rule coverage, which weighs each rule by the size of the match set, is high because the default route matches the vast majority of the destination IP address space (anything that is not covered by a more specific rule). This data supports the engineers' intuition to prioritize default routes in their testing.

Low fractional rule coverage became the catalyst for identifying testing gaps. Focusing on forwarding rules that Yardstick reported as untested, we identified three categories of routes.

(1) *Internal routes.* Each device has many prefixes to destinations that are internal to the region. These include prefixes for hosts connected to ToRs and prefixes of loopback interfaces on all routers.
(2) *Connected routes.* Each device has connected routes that correspond to its physical and aggregated interfaces, which have statically configured /31 (IPv4) and /126 (IPv6) prefixes. Because these prefixes are used for point-to-point connections only, they are not redistributed into the global eBGP routing protocol. Yardstick flagged that none of these connected routes, and many of their associated interfaces, were untested.
(3) *Wide-area routes.* Yardstick revealed that routers in the upper layers of each data center had particularly low rule coverage. Upon further investigation, we found that these rules were for routes learned from the wide-area network, which are advertised to the regional hub and spine layers but are not leaked into lower layers.

At first blush, it may appear that a system like Yardstick is an overkill for identifying such gaps–engineers should have known about them based on their knowledge of the network and the test suite. But the real-world aspects of this challenge are worth noting. First, the engineers who originally developed the test suite can be different from those who are now responsible for maintaining and improving it. It can be difficult for the latter group to look at the old test code and judge what the tests are not covering. Second, real networks lose their original design simplicity and symmetry over time. This evolution makes it difficult to reason about network structure in one's head, to accurately identify which components are tested and which ones are not. The coverage information that Yardstick provides from various perspectives is thus key to comprehensively and reliably identifying testing gaps.

Yardstick also makes it easy to focus one's efforts on the most productive kind of test development—the creation new tests that provably improve coverage—rather than on development of redundant tests that do little to find additional errors in networks. We discuss the new tests developed for our network next.

7.3 Toward a High-coverage Test Suite

After identifying the testing gaps above, the engineers authored two new tests for the network.

(1) *InternalRouteCheck.* This test validates that all prefixes that originate within the datacenter (*i.e.*, the internal destinations) are forwarded through and only through the full set of topological shortest paths. The design of the network is such that internal destinations are routed along shortest paths

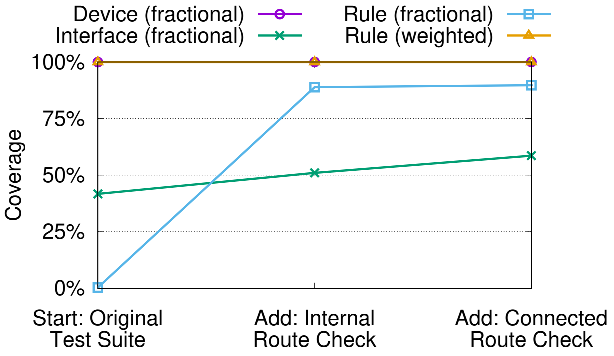


Figure 7: Coverage improvement with test suite iterations.

and there are many such paths. The test is implemented as a local symbolic test that uses RCDC's idea [19] of decomposing an end-to-end invariant into local forwarding contracts that dictate the next hops for a prefix at a device. Suppose a device v originates a set of prefixes $\{p_v\}$, which include host subnets and loopback addresses. To compute the local contracts for $\{p_v\}$, the test first perform a breadth-first search from v to compute shortest distances from all other devices. If the device v' is d-hops away from v, it should then forward $\{p_v\}$ to all its neighbors with distance of $d-1$ to v.

(2) *ConnectedRouteCheck.* This test validates that both ends of a physical link have the connected route to the assigned /31 and /126 prefixes. It is a state-inspection test.

The engineers have yet to define a test for wide-area routes, the third gap that Yardstick helped identify. The challenge is that there is not yet any specification of the routes to expect from the wide-area network.

The two tests above are now deployed.[5] The coverage of these tests is shown in Figures 6b and 6c. InternalRouteCheck covers over 90% of the rules on ToR and aggregation routers, and around 50% on spine and regional hub routers. Its impact differs across router types based on the fraction of internal routes that a type contains. ConnectedRouteCheck covers nearly 100% of the interfaces on all routers except for ToRs.

The coverage for the final test suite, after adding these two tests to the original test suite, is shown in Figure 6d. The coverage is substantially higher than before. But fractional rule coverage on spine routers and regional hubs is only around 50% (because of untested wide area routes). Fractional interface coverage on ToR routers is only 25%. This coverage level is similar to that of the original test suite and of the two new tests individually, indicating that all tests are covering the same set of interfaces. We discovered that host-facing interfaces are not being tested, and as a result, will be developing another new test for these interfaces soon.

Figure 7 summarizes the coverage improvement across all devices during the course of our study. Within the first month of its deployment, Yardstick helped improve rule coverage by 89% and interface coverage by 17%.

8 PERFORMANCE EVALUATION

We conducted controlled experiments to benchmark the performance of Yardstick along two measures of interest: *i*) the overhead of tracking coverage when the tests are running; and *ii*) the time it takes to compute the metrics after the tests are done. We generate synthetic fat-tree networks [1] of different sizes by varying the topology parameter k between 8 and 88, which generates networks of up to 9680 routers. Each ToR has one hosted prefix, and network routing functions as described in §7.1.

All experiments were performed on a desktop PC with an 8C8T Intel CPU running at 4.9 GHz and 16 GB of DRAM.

8.1 Overhead of coverage tracking

We measure the overhead of coverage tracking by running tests with and without tracking enabled. We consider four types of tests:

- *DefaultRouteCheck* is the state-inspection test mentioned earlier. It determines whether each switch has a default route that forwards to higher-layer neighbors.
- *ToRReachability* is an end-to-end symbolic test. It checks that all packets that originate at a ToR with a destination IP address in the hosted prefix of another ToR can reach the correct ToR.
- *ToRContract* is a local symbolic test. It checks the same invariants as ToRReachability, but does so by decomposing the invariant into a local forwarding contract for each router. ToRContract is a subset of RCDC [19].
- *ToRPingmesh* is an end-to-end concrete test. It checks the same invariants as as *ToRReachability*, but samples a single address from each prefix instead of reasoning about all packets. The idea of testing ToR pairs using concrete packets is drawn from Pingmesh [14].

Figure 8 shows the time to execute each test with and without coverage tracking enabled. We see that the overhead of coverage tracking is small. The worst case absolute time overhead is 54 seconds, which occurs for the ToRReachability test in the largest topology. In this case, the baseline (coverage disabled) test time is 1967 seconds, and thus the relative overhead is only 2.8%.

The worst case relative overhead is 52%, which occurs for DefaultRouteCheck in the topology with 6480 nodes. In this case, the baseline test time is only 0.79 seconds. (State-inspection tests are lightweight.) Across all cases where the baseline test takes over a minute, the relative overhead of tracking coverage is under 10%.

8.2 Performance of coverage computation

After the tests finish, Yardstick computes coverage metrics using the coverage trace and network state. Figure 9 shows the computation time for different components (when computed by itself) as a function of network size. The coverage trace for this experiment is from the tests in the previous section. We show results for fractional averages; performance is similar for other aggregations.

We see that local metrics (*i.e.*, device, interface, and rule coverage) are reasonably fast to compute. Each metric is computed in less than 90 seconds, even on the largest topology. We also find that

[5] These tests did not identify unique bugs during the study. All discovered bugs were shallow and were flagged by the DefaultRouteCheck as well. However, one cannot rely on bugs being always shallow. Yardstick enabled the engineers to add tests that improve coverage, which improves the chances of finding additional bugs and increases confidence in the correctness of network changes.

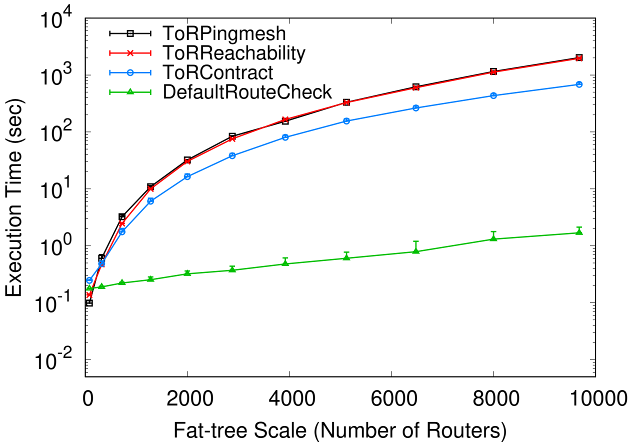


Figure 8: Overhead of coverage tracking. The lines show the test execution time without coverage tracking, and the error bars show the time with coverage tracking.

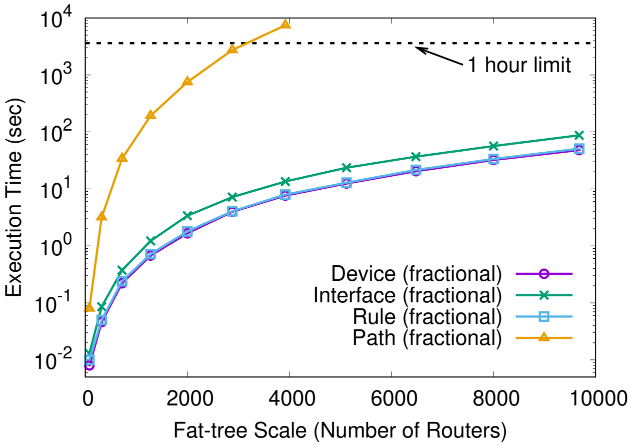


Figure 9: Time to compute coverage metrics.

because a fair bit of processing is shared among these metrics (*e.g.*, computing match sets of rules), computing all of them together takes only 91 seconds (not shown in the graph).

Path coverage, on the other hand, is computationally expensive. It takes 45 minutes on the 2880-node network. On larger networks, its execution time exceeds the 1-hour timeout we used for these experiments. This metric is expensive because it requires iterating over all possible paths in the topology. As our networks use multipath routing, doing so becomes prohibitive beyond a certain point.

Network engineers that we have talked to mentioned that Yardstick is efficient enough to be useful in practice. Coverage metrics are not expected to change significantly at short time scales unless the network state or the test suite changes significantly. The engineers thus mentioned computing path-based expensive metrics once a day, while relying on the local metrics to more quickly catch regressions in testing.

9 RELATED WORK

Our work builds on two distinct lines of research. The first line of research involves network testing and verification tools [11, 14, 15, 19, 21, 22, 32, 33]. These tools have developed a range of techniques for scalable analysis of network data and control planes. We have borrowed some of these ideas (*e.g.*, the use of BDDs) to compute coverage metrics efficiently. However, the goals of our research are different from, and complementary to, this past work: we have developed metrics that systematically quantify how well verification and testing tools have been put to use by network engineers in practical industrial settings.

Second, we borrow from the software domain the idea of using coverage metrics to quantify test suite quality and reveal testing gaps. There, many metrics have been developed over the years to help software engineers [5, 12, 16]. Our network coverage metrics are specialized for operation over network forwarding state.

We share with ATPG [34] the goal of improving network testing. ATPG crafts test packets that exercise each rule in the network forwarding state to validate that routers indeed forward the packet as per their state. It thus improves testing that aims to find bugs in device software and hardware responsible for forwarding packets. In contrast, we improve testing that aims to find bugs in the forwarding state itself. In service of this goal, we also develop several notions of coverage beyond rule coverage.

We introduced the idea of using coverage metrics to improve the use of network verification in a position paper [7]. That paper did not develop a computational basis or a system to compute coverage metrics, nor did it report on experience of using coverage metrics to improve test infrastructure of large-scale, industrial networks.

10 CONCLUSION

We described a framework to define and compute network coverage metrics, to help engineers judge and improve the quality of their test suites. To be able to compute a range of metrics using a diverse set of tests, it is based on decomposing both the state exercised by testing and network components into atomic testable units. We built our system Yardstick based on this framework and deployed it in Microsoft Azure. Within the first month of deployment in one of the production networks, Yardstick helped expand testing to 17% more network interfaces and 89% more rules.

These results notwithstanding, we believe we have barely scratched the surface of network coverage metrics. For software, the development of coverage metrics and usable systems has been a multidecade journey that still continues. We expect networking to take a similar journey, with researchers developing increasingly sophisticated metrics for a range of settings. The framework developed in this paper can provide a useful starting point for that journey.

ACKNOWLEDGMENTS

We thank the anonymous SIGCOMM '21 reviewers and our shepherd, Aman Shaikh, for feedback on earlier versions of this paper. This work was supported in part by NSF grants CNS-2007073 and CNS-1703493.

This work does not raise any ethical issues.

REFERENCES

[1] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. 2008. A Scalable, Commodity Data Center Network Architecture. In *Proceedings of SIGCOMM '08*. ACM, 63–74.

[2] Paul Ammann and Jeff Offutt. 2016. *Introduction to Software Testing.* Cambridge University Press.

[3] Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. 2014. NetKAT: Semantic Foundations for Networks. In *Proceedings of POPL '14*. ACM, 113–126.

[4] Mae Anderson. 2014. *Time Warner Cable Says Outages Largely Resolved.* Retrieved June 23, 2021 from http://www.seattletimes.com/business/time-warner-cable-says-outages-largely-resolved

[5] James H Andrews, Lionel C Briand, Yvan Labiche, and Akbar Siami Namin. 2006. Using mutation analysis for assessing and comparing testing coverage criteria. *IEEE Transactions on Software Engineering* 32, 8 (2006), 608–624.

[6] John Backes, Sam Bayless, Byron Cook, Catherine Dodge, Andrew Gacek, Alan J Hu, Temesghen Kahsai, Bill Kocik, Evgenii Kotelnikov, Jure Kukovec, et al. 2019. Reachability analysis for AWS-based networks. In *International Conference on Computer Aided Verification*. Springer, 231–241.

[7] Ryan Beckett and Ratul Mahajan. 2019. Putting Network Verification to Good Use. In *Proceedings of HotNets '19*. ACM, 77–84.

[8] Larry Brader, Howie Hilliker, and Alan Wills. 2013. *Testing for Continuous Delivery with Visual Studio 2012.* Microsoft.

[9] Mosharaf Chowdhury, Yuan Zhong, and Ion Stoica. 2014. Efficient coflow scheduling with varys. In *Proceedings of SIGCOMM '14*. ACM, 443–454.

[10] Codecov. 2021. *CodeCov: The leading code coverage solution.* Retrieved June 23, 2021 from https://about.codecov.io/

[11] Ari Fogel, Stanley Fung, Luis Pedrosa, Meg Walraed-Sullivan, Ramesh Govindan, Ratul Mahajan, and Todd Millstein. 2015. A General Approach to Network Configuration Analysis. In *Proceedings of NSDI 15*. USENIX Association, 469–483.

[12] Milos Gligoric, Alex Groce, Chaoqiang Zhang, Rohan Sharma, Mohammad Amin Alipour, and Darko Marinov. 2013. Comparing Non-Adequate Test Suites Using Coverage Criteria. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis (ISSTA 2013)*. 302–313.

[13] 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 SIGCOMM '09*. ACM, 51–62.

[14] Chuanxiong Guo, Lihua Yuan, Dong Xiang, Yingnong Dang, Ray Huang, Dave Maltz, Zhaoyi Liu, Vin Wang, Bin Pang, Hua Chen, Zhi-Wei Lin, and Varugis Kurien. 2015. Pingmesh: A Large-Scale System for Data Center Network Latency Measurement and Analysis. In *Proceedings of SIGCOMM '15*. ACM, 139–152.

[15] Alex Horn, Ali Kheradmand, and Mukul Prasad. 2017. Delta-net: Real-time Network Verification Using Atoms. In *Proceedings of NSDI 17*. USENIX Association, 735–749.

[16] Monica Hutchins, Herb Foster, Tarak Goradia, and Thomas Ostrand. 1994. Experiments on the effectiveness of dataflow-and control-flow-based test adequacy criteria. In *Proceedings of 16th International conference on Software engineering*. IEEE, 191–200.

[17] Marko Ivanković, Goran Petrović, René Just, and Gordon Fraser. 2019. Code coverage at Google. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 955–963.

[18] Karthick Jayaraman, Nikolaj Bjørner, Geoff Outhred, and Charlie Kaufman. 2014. *Automated analysis and debugging of network connectivity policies.* Technical Report. 1–11 pages. https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/secguru.pdf

[19] 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, and Parag Sharma. 2019. Validating Datacenters at Scale. In *Proceedings of SIGCOMM '19*. ACM, 200–213.

[20] Peyman Kazemian, Michael Chang, Hongyi Zeng, George Varghese, Nick McKeown, and Scott Whyte. 2013. Real Time Network Policy Checking Using Header Space Analysis. In *Proceedings of NSDI 13*. USENIX Association, 99–111.

[21] Peyman Kazemian, George Varghese, and Nick McKeown. 2012. Header Space Analysis: Static Checking for Networks. In *Proceedings of NSDI 12*. USENIX Association, 113–126.

[22] Ahmed Khurshid, Xuan Zou, Wenxuan Zhou, Matthew Caesar, and P Brighten Godfrey. 2013. Veriflow: Verifying network-wide invariants in real time. In *Proceedings of NSDI 13*. USENIX Association, 15–27.

[23] Petr Lapukhov, Ariff Premji, and Jon Mitchell. 2016. Use of BGP for routing in large-scale data centers. *Internet Requests for Comments RFC Editor RFC* 7938 (2016).

[24] 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 SOSP '17*. ACM, 599–613.

[25] Nuno P Lopes and Andrey Rybalchenko. 2019. Fast bgp simulation of large datacenters. In *International Conference on Verification, Model Checking, and Abstract Interpretation*. Springer, 386–408.

[26] Glenford J. Myers, Corey Sandler, and Tom Badgett. 2012. *The art of software testing* (3rd ed ed.). John Wiley & Sons.

[27] Steve Ragan. 2016. *BGP errors are to blame for Monday's Twitter outage, not DDoS attacks.* Retrieved June 23, 2021 from https://www.csoonline.com/article/3138934/security/bgp-errors-are-to-blame-for-monday-s-twitter-outage-not-ddos-attacks.html

[28] Deon Roberts. 2018. *It's been a week and customers are still mad at BB&T.* Retrieved June 23, 2021 from https://www.charlotteobserver.com/news/business/banking/article202616124.html

[29] Yevgeniy Sverdlik. 2017. *United Says IT Outage Resolved, Dozen Flights Canceled Monday.* Retrieved June 23, 2021 from https://www.datacenterknowledge.com/archives/2017/01/23/united-says-it-outage-resolved-dozen-flights-canceled-monday

[30] Bingchuan Tian, Xinyi Zhang, Ennan Zhai, Hongqiang Harry Liu, Qiaobo Ye, Chunsheng Wang, Xin Wu, Zhiming Ji, Yihong Sang, Ming Zhang, Da Yu, Chen Tian, Haitao Zheng, and Ben Y. Zhao. 2019. Safely and Automatically Updating In-Network ACL Configurations with Intent Language. In *Proceedings of SIGCOMM '19*. ACM, 214–226.

[31] Zach Whittaker. 2020. *T-Mobile hit by phone calling, text message outage.* Retrieved June 23, 2021 from https://techcrunch.com/2020/06/15/t-mobile-calling-outage/

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

[33] Fangdan Ye, Da Yu, Ennan Zhai, Hongqiang Harry Liu, Bingchuan Tian, Qiaobo Ye, Chunsheng Wang, Xin Wu, Tianchen Guo, Cheng Jin, Duncheng She, Qing Ma, Biao Cheng, Hui Xu, Ming Zhang, Zhiliang Wang, and Rodrigo Fonseca. 2020. Accuracy, Scalability, Coverage: A Practical Configuration Verifier on a Global WAN. In *Proceedings of SIGCOMM '20*. ACM, 599–614.

[34] Hongyi Zeng, Peyman Kazemian, George Varghese, and Nick McKeown. 2012. Automatic test packet generation. In *Proceedings of the 8th international conference on Emerging networking experiments and technologies*. 241–252.

[35] Hongyi Zeng, Ratul Mahajan, Nick McKeown, George Varghese, Lihua Yuan, and Ming Zhang. 2015. *Measuring and Troubleshooting Large Operational Multipath Networks with Gray Box Testing.* Technical Report MSR-TR-2015-55 (Microsoft Research).

[36] 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 *Proceedings of NSDI 14*. USENIX Association, 87–99.