Test Coverage for Network Configurations

Xieyang Xu and Weixin Deng, *University of Washington;* Ryan Beckett, *Microsoft;*
Ratul Mahajan, *University of Washington;* David Walker, *Princeton University*

https://www.usenix.org/conference/nsdi23/presentation/xu

This paper is included in the
Proceedings of the 20th USENIX Symposium on
Networked Systems Design and Implementation.

April 17–19, 2023 • Boston, MA, USA

978-1-939133-33-5

Open access to the Proceedings of the
20th USENIX Symposium on Networked
Systems Design and Implementation
is sponsored by

جامعة الملك عبدالله
للعلوم والتقنية
King Abdullah University of
Science and Technology

Test Coverage for Network Configurations

Xieyang Xu[1] Weixin Deng[1] Ryan Beckett[2] Ratul Mahajan[1] David Walker[3]

[1]*University of Washington* [2]*Microsoft* [3]*Princeton University*

Abstract

We develop NetCov, the first tool to reveal which network configuration lines are tested by a suite of network tests. It helps network engineers improve test suites and thus increase network reliability. A key challenge in developing a tool like NetCov is that many network tests test the data plane instead of testing the configurations (control plane) directly. We must be able to efficiently infer which configuration elements contribute to tested data plane elements, even when such contributions are non-local (on remote devices) or non-deterministic. NetCov uses an information flow graph based model that precisely captures various forms of contributions and a scalable method to infer contributions. Using NetCov, we show that an existing test suite for Internet2, a nation-wide backbone network in the USA, covers only 26% of the configuration lines. The feedback from NetCov makes it easy to define new tests that improve coverage. For Internet2, adding just three such tests covers an additional 17% of the lines.

1 Introduction

As critical infrastructure, networks must be highly reliable but, unfortunately, network outages are common. A primary culprit is networks' reliance on complex, low-level configuration that dictates how routers select best paths and forward traffic. Day-to-day updates to network configuration are error-prone, leading to outages that knock off important online services (e.g., banking), ground airplanes, and disable critical communication (e.g., emergency calls) [3, 33, 34, 39, 45].

To improve network reliability, automatic testing and verification of configurations is becoming commonplace. Today, network operators have at their disposal many tools with increasing sophistication that can scale to large networks and check various aspects of network behavior [5, 23, 40, 49, 51].

However, using such tools is not sufficient by itself; one must also use them *effectively*. Outages can occur despite automated testing when the test suite is poor and does not cover key aspects of network configuration. This was the case with the massive Facebook outage during which Facebook, WhatsApp, Instagram, and Oculus were unavailable for six hours [35]. Current tools have pushed the limits of *what can be tested* but left open the question of *what needs to be tested*.

Without tool support, it is difficult for engineers to know if they are effectively testing network configurations. In industrial networks with possibly millions of lines of configurations, engineers' understanding of network behavior and dependencies is necessarily incomplete. It is even harder to update an existing test suite after the network evolves because the engineers likely do not know what the old test suite is or is not testing for the updated network.

Recent work has proposed data plane coverage [47] to reveal testing gaps. It shows which data plane elements, such as forwarding rules, are exercised by a test suite. However, well-tested data plane does not imply well-tested configurations. Data plane elements are the output of network's configurations (which define its control plane) and the current operating environment (failures, external routing information). Testing a given data plane only tests configuration elements that are exercised in that particular environment. Other configuration elements are not tested. We demonstrate this empirically via a scenario where testing *all* data plane elements leaves over half of configuration lines untested.

We develop *configuration coverage* to provide comprehensive and precise feedback to network engineers on test suite quality. Our goal is to identify exactly which configuration lines are tested and which ones are not. We want to consider all configuration elements, not only those that contribute to the current data plane. Revealing exactly which lines are untested helps improve tests—add tests that target untested lines—which in turn can improve network reliability. This is similar to how code coverage tools help improve tests and software reliability [9, 11, 22].

A major challenge we face is that many network tests do not exercise configurations directly. Instead, they reason about the data plane elements produced by configurations. We need to infer the configuration elements that contribute to the tested data plane elements. This inference is complicated because contributions can be non-local and non-deterministic. In a distributed control plane, a piece of tested routing information may have been propagated and transformed multiple times along its path, and both local and non-local configurations may have contributed to its existence. For example, the path attributes of a BGP route is shaped by routing policies on each and every hop that it traverses. Further, not all contributions are deterministic. For instance, any one of possibly multiple sub-prefixes can lead to the route of an aggregate prefix. We must scalably account for local and non-local contributions and for non-deterministic contributions.

Our solution is to model the contribution between configuration elements and data plane elements as an *information*

flow graph. An IFG is a directed acyclic graph (DAG) where vertices denote network elements and edges denote contributions. In addition to direct contributions from configuration elements to data plane elements, we also model contributions between data plane elements (from predecessors to successors). For instance, a BGP route contributes to the BGP message that derived from it. Indirect contributions are thus modeled by multi-hop paths in the DAG. When contributions exhibit non-determinism, we use special *disjunctive* nodes to organize possible DAG paths that may contribute to a given data plane element.

We build a tool called NetCov based on this model. It annotates which configuration lines and logical elements are tested by a given test suite and produces aggregated coverage statistics. To efficiently map tested data plane element to the set of contributing configuration elements, it materializes the IFG lazily, instead of tracking contributions proactively, during data plane generation. This design avoids the cost to compute and store contributions for transient or untested data plane elements. NetCov is open-sourced on GitHub [30].

We evaluate NetCov on Internet2, a nation-wide backbone network in the USA, and on synthetic data center networks. We show that test suites proposed in prior work can have poor coverage. The three tests proposed by Bagpipe [44] covered only 26% of the configuration lines of Internet2. We also show how surfacing untested configuration elements suggests new tests that improve coverage. By adding just three such tests to the Internet2 test suite based on NetCov's feedback, we could improve coverage to 43%, and more similar tests can be added to further increase coverage. NetCov performs reasonably well. The time to compute coverage is 1.2 hours for the largest network that we study, which has over 2 million forwarding rules. This time is an order of magnitude less than the time to execute tests.

Stepping back, we note that networking is not alone in its reliance on configuration. Today, a lot of infrastructure and distributed applications are deployed by composing existing components using configuration (e.g., infrastructure deployment using Terraform, and application deployment using containers and service meshes). These configurations are central to correct behavior, which is why there is an intense focus on testing them properly [21, 38, 43]. As for networks, there are no tools to help engineers discover how well the configurations are tested. The techniques developed in our work, the IFG-based contribution tracking and its lazy traversal, can provide a starting point toward better testing of infrastructure and distributed application configuration as well.

2 Background on Network Testing

In networks with distributed control planes, each device runs one or more routing protocol (e.g., BGP, OSPF) instances. Each instance exchanges routing messages with its neighboring instances. Routing messages contain attributes of paths that the sender is using to various destinations. A routing instance may learn multiple paths to the same destination via different neighbors. It selects the best one (or multiple best ones if multipath routing is enabled) based on its policy and stores that path in its protocol RIB (routing information base). Multiple routing protocol instances on a device may have best paths to the same destination. The device selects the best one(s) based on the relative preference of the protocols and stores the selection in its main RIB. Information in the main RIB is used to forward packets.[1]

Network engineers can control many aspects of the computation above using device configuration. This includes the routing protocol instances that are running; the peering between instances; the destination prefixes that are announced by each routing protocol instance; how routing messages are transformed prior to sending (export policy) and upon reception (import policy); and the preference function for best path selection. Naturally, thus, how the network forwards packets is intimately dependent on device configurations.

Given the importance of configurations to correct network behavior, network engineers use automatic testing to find bugs and gain confidence in their correctness. Network tests come in two flavors. *Data plane tests* analyze the computed data plane state (*i.e.*, RIBs), e.g., checking that node A can reach B and that route to a particular destination is present at node C. *Control plane tests* directly analyze device configuration, e.g., checking that the import policy blocks routing messages for private address space (such as 10.0.0.0/8) and BGP peerings are correctly configured.

3 Configuration Coverage: Overview

Network engineers today create data and control plane tests based on past outages and their knowledge of which behaviors are important to test. There are no tools to provide feedback on how well they are testing configurations and which aspects of the configuration are untested. We aim to build such a tool. Given the complexity of real-world networks, it is difficult for humans to know if they have covered all important elements of configurations. As with software, high coverage is necessary but not sufficient for a good test suite. In addition to exercising all key behaviors, the tests must also properly assert that those behaviors match intent. This latter task is not our focus.

Our goal is to reveal which elements of the network configuration are covered by a suite of data and control plane tests. Before discussing our approach, we define what it means for a configuration element to be covered.

[1] In reality, for fast forwarding, routers have a forwarding information base (FIB), which maps each main RIB destination to its outgoing interface, by recursively resolving next hop information (which may be an IP address). The difference between main RIB and FIB is not material for our work, and we use the term main RIB for the table that has forwarding information.

3.1 Defining coverage

We deem a configuration element to be covered if it *i*) is tested directly by a control plane test; or *ii*) contributes to the production of a data plane state element (i.e., an entry in the protocol or main RIB) tested by a data plane test. For now, assume that contributions are deterministic. We discuss non-deterministic contributions in the next section.

Figure 1 illustrates configuration coverage as a result of a data plane test. It shows parts of the two routers' configuration. R1's configuration defines one interface (Lines 1-2) and one BGP peer (192.168.1.2, which is R2's address), and it specifies the import and export policy to use. The import policy (R2-to-R1 at Lines 6-11) denies routing messages for a particular prefix and sets the preference for another.

R2's configuration defines two interfaces, a BGP peer (R1) and routing policies. At Line 13, it states that the prefix 10.10.1.0/24 should be announced to BGP peers *iff* it is in the main RIB.[2] In our example, 10.10.1.0/24 will be in the main RIB as it corresponds to the eth1's prefix. (Address statements like Line 4 encode the IP address and prefix length. For eth1, given the address 10.10.1.1 and prefix length of 24, the prefix is 10.10.1.0/24.) Routers add interface prefixes to the "connected" protocol RIB, from where those prefixes can enter the main RIB. The resulting RIBs on the two routers are shown in the figure. Each entry includes the next hop and source routing protocol ("conn" = connected).

Suppose the entry for 10.10.1.0/24 at R1 was tested by a data plane test. The covered configuration elements are highlighted. On R1, the BGP peer configuration and import policy binding (Lines 3-4) are covered because the tested entry came via that peering and passed through that policy. Parts of the routing policy R2-to-R1 relevant to the tested state (Lines 6, 9-11) are also covered. The interface definition (Lines 1-2) is covered because it enables the BGP peering to be established. In contrast, the export policy R1-to-R2 and unexercised parts of R2-to-R1 (Lines 7-8) are not covered.

There are covered configuration elements at R2 as well. These include the interface definitions—eth0 enables the BGP edge and 10.10.1.0/24 was announced due to eth1—and BGP peering, the export policy, and the BGP network statement.

Alternative definitions of coverage. One may consider an alternative definition of coverage that disregards non-local configuration elements. But we posit that including non-local elements is more meaningful. These elements, such as the BGP network statement on R2's Line 13, are just as key to the existence of 10.10.1.0/24 at R1 as the local elements.

Another definition of coverage is based on mutation [4]: a configuration line is deemed covered if its mutation alters the test result. Compared to the definition of coverage we adopt, mutation-based coverage will report an additional class of configuration elements as covered—configuration elements

[2]Different router vendors have different semantics for BGP network statements. We are assuming Cisco semantics.

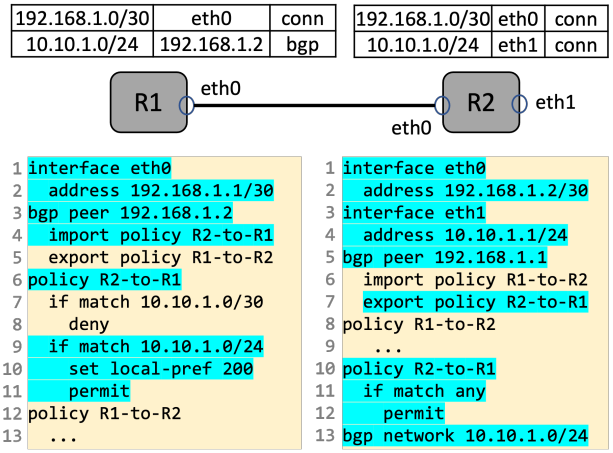


Figure 1: An example network with routing tables and configurations. The highlighted configuration lines are covered when the route to 10.10.1.0/24 is tested at R1.

that de-prioritize (or reject) the competitors of the tested data plane element. Mutation-based coverage tends to be significantly harder to compute [24], and its results can be hard to interpret. In developing the first tool in this space, we decided to focus on a simpler, more direct definition of coverage. We will explore more sophisticated definitions in the future.

3.2 Our approach

While it is straightforward to identify configuration elements covered by a control plane test, it is not so for data plane tests. Data plane tests analyze the "output" of the control plane, and we need a scalable way to compute which configuration elements contributed to tested data plane state. The relationship between these inputs and outputs is complex. How a particular RIB entry comes about relies on many configuration elements across multiple devices. The need to map tested outputs to input space sets computation of configuration coverage apart from data plane coverage and software coverage, for both of which the coverage domain is the same as test domain.

To motivate our approach to solving this problem, let us first sketch two strawman approaches. One potential approach is to express control plane computation declaratively, e.g., in Datalog. This enables identification of contributing inputs for a given output using a form of backward-reasoning [46, 52]. However, network control plane computations can be quite complex (e.g., non-monotonic behaviors [16,36]). While declarative encodings may work in special cases [27], it is generally hard to get high-fidelty, performant encodings. That is why most control plane analysis tools use an imperative approach [12, 31, 32, 49].[3]

[3]Batfish [12], a widely used control plane analysis tool, originally used Datalog to encode network control planes but switched to imperative simulations due to expressiveness and performance challenges.

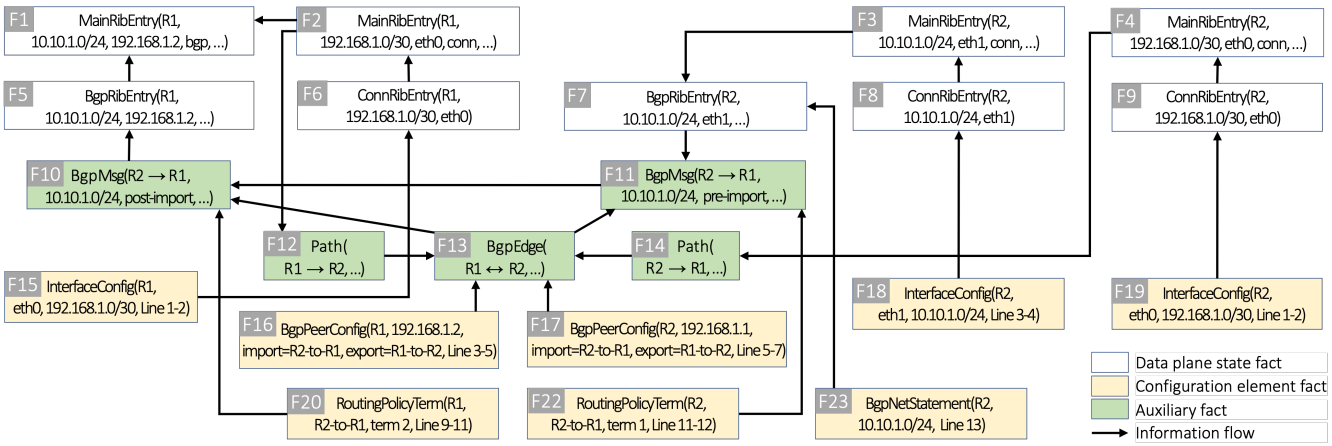


Figure 2: Subset of the IFG for the Figure 1 example. It tracks configuration elements that contribute to the tested RIB entry (F1).

Another potential approach is to use simulation-based forward reasoning, i.e., simulate the control plane (imperatively) and track which configuration elements feed into each part of the data plane state. However, this approach has scalability limitations. Network simulation is time and memory intensive [12, 32, 49], and it will become significantly worse if it needed to track all necessary information along each hop.

Our approach is based on two observations. First, for the purposes of computing coverage, we do not need a full computational model of the control plane. We need to only track which configuration elements contribute to tested data plane state (i.e., taint analysis [41]), not the exact input-output relationship; and we need to reason only about the stable state (i.e., the the of devices once they have settled on best paths), not the transient states. Data plane testing [19, 23, 25, 26, 48] assumes that the analyzed state is stable. Our second observation is that the stable state contains enough information for us to infer contributions of configuration elements after the fact, based on the semantics of the control plane. This inference is vastly cheaper than tracking contributions towards all data plane state entries, independent of whether they are tested.

To model contributions to the stable state, we use an *information flow graph* (IFG). Figure 2 shows a subset of the IFG for the example in Figure 1. Each node is a *fact* and arrows denote information flow from the tail to head. IFGs have three types of facts: *i*) data plane state, *ii*) configuration elements, and *iii*) auxiliary facts that capture intermediate dependencies between data plane state and configuration elements.

The main RIB entry 10.10.1.0/24 at R1 (F1) is derived from the corresponding BGP RIB entry (F5), which in turn is derived from the BGP message from R2 (F10). This message exists because of the BGP edge between R1 and R2 (F13), the source message sent by R2 (F11), and the relevant configuration element within import policy (F20). R2 sent the BGP message because of the same BGP edge (F13), its export policy elements (F22), and the BGP RIB entry (F7). This BGP RIB entry exists because of the configuration element (F23) and the RIB entry (F3), which exists because of the connected route (F8). The BGP edge (F13) exists because of the configuration elements that define the peering (F16, F17) and paths between R2 and R1 that enable the BGP session to be established. The paths depend on the RIB entries (F2 and F4, respectively), the contributions to which can be similarly traced. In this manner, the IFG captures all configuration elements that led to the tested RIB entry (F1).

We do not track IFG dependencies proactively but infer them on-demand based on control plane semantics, using a mix of backward-forward reasoning. Backward inference infers the parent (tail) of the edge from its child (head). The information in child nodes is not enough to fully recover the parent nodes, but is often enough to select them from the known stable state. For instance, we can compute the BGP RIB entry F5 from the main RIB entry F1—the main RIB entry indicates that its source routing protocol is BGP, and we thus look up the BGP RIB for 10.10.1.0/24.

Lookup-based inference does not always work. For instance, given a BGP message which has passed through an import policy, we cannot compute backwards which terms of the import policy were exercised (F10 ← F20). Another parent of F10, the pre-import BGP message (F11) cannot be looked up either because it is not part of the input and needs to be computed on-the-fly. To address these limitations, we combine backward and forward inference. When a parent can not be directly looked up, we first look up the prerequisites of the parent. For instance, we can look up F7 based on F10. Next, we use targeted simulations to compute non-existing facts and to select relevant facts exercised in a control plane process or data plane process. For instance, given the BGP route at R2 (F7), we simulate its processing through the export policy, which allows us to derive the pre-import BGP message (F11) and find the policy term exercised during the export process (F22). Once F11 is computed, we conduct another targeted

simulation to discover the policy term exercised in the import process (F20). Unlike a full control plane simulation, these targeted simulations are fast. They have limited scope (e.g., best path selection is not simulated) and are done only for messages of interest, not all messages.

By combining backward and forward inference, atop the stable state IFG, we can scalably discover all covered configuration elements. We describe this approach in detail next.

4 Design of NetCov

NetCov takes as input configuration files, data plane state (protocol RIBs, main RIB and active routing edges) of the network. The data plane state may be pulled from live network or produced by control plane analysis tools [12, 32, 49]. In addition, NetCov takes as input what is tested: data plane entries that are tested by data plane tests, and configuration elements that are tested by control plane tests. This information is produced by network testing tools [12, 47].

Based on these inputs, NetCov computes which configuration elements are covered. The core of this computation efficiently mapping a data plane fact to configuration elements that contribute to it. We describe this computation next.

4.1 Information flow model

IFGs are directed acyclic graphs whose nodes denote network *facts* and edges denote information flow between facts. Table 1 shows the types of network facts modeled by NetCov and the information flow between different types. Data plane state has three subtypes: main RIB entries, protocol RIB entries, and access control list (ACL) entries.

Auxiliary facts have three subtypes: routing edges, routing messages, and paths of routing messages. These facts are not strictly necessary, but they help create a compact IFG and speed up graph walking. For instance, the routing messages of many protocol RIB entries depend on the same path which in turn may depend on many main RIB entries. Adding an explicit fact for the path avoids the need to add all pairs of edges between routing messages and main RIB entries.

In our model, the auxiliary facts for routing messages represent messages between routing protocol instances across devices as well as within a device, i.e., redistribution [10]. This uniform treatment is a modeling convenience. In reality, explicit messages are not exchanged during redistribution (though redistribution is subject to routing policies akin to messages between cross-device routing instances).

The last column of Table 1 shows how information flows among different types of facts. A main RIB entry stems from a protocol RIB entry and optionally another main RIB entry (when its next hop is an IP address whose corresponding output interface needs further resolution). A protocol RIB fact stems from a routing message (for protocols such as BGP), a configuration element (for connected interfaces and static routes), a main RIB entry accompanied with a configuration element (such as when a BGP network statement populates a main RIB entry into BGP RIB) or a set of RIB entries accompanied with a configuration element (for aggregate routes). ACLs facts stem from configuration facts and have no other dependencies. Routing messages stem from a RIB fact or another message (e.g., post-import-policy message depends on pre-import-policy message), and they also depend on routing edges and routing policy configurations. Inter-device routing edges stem from paths that enable sessions to be established and configuration facts that define peerings; Intra-device routing edges stem from configuration facts that define redistribution. Finally, path facts depends on main RIB facts and ACL facts that impact routing traffic along the way.

Network fact		**Information flow**
Configuration element (c)		None
Data plane state	Main RIB entry (f)	$f_i \leftarrow r_j$ $f_i \leftarrow r_j, f_k$
	Protocol RIB entry (r)	$r_i \leftarrow m_j$ $r_i \leftarrow c_j$ $r_i \leftarrow f_j, c_k$ $r_i \leftarrow \{r_{j_1}, ...\}, c_k$
	ACL entry (a)	$a_i \leftarrow \{c_{i_1}, ...\}$
Auxiliary	Routing message (m)	$m_i \leftarrow r_j, e_k, \{c_{l_1}, ...\}$ $m_i \leftarrow m_j, e_k, \{c_{l_1}, ...\}$
	Routing edge (e)	$e_i \leftarrow \{c_{j_1}, ...\}$ $e_i \leftarrow \{c_{j_1}, ...\}, \{p_{k_1}, ...\}$
	Path (p)	$p_i \leftarrow \{f_{j_1}, ...\}, \{a_{k_1}, ...\}$

Table 1: Information flow model: Types of facts and all possible information flows for each type. $\{t, ...\}$ denotes a set of facts.

For correct computation of coverage, the IFG model must be sound and realizable. Soundness means that it includes all relevant dependencies (per control plane semantics) and no more. Realizable means parents (tails) along all information flow edges can always be inferred, via lookup or simulation or a mix. Our model is sound to our knowledge; and that we are able to use it to compute coverage, using the framework described next, points to its realizability.

4.2 Inferring the IFG on demand

Based on the information flow model, NetCov uses a backward-forward inference framework to lazily materialize the IFG from any set of facts whose coverage need to be tracked. The framework is abstracted using a set of *inference rules* and an iterative construction algorithm. Each inference rule is function that takes a materialized IFG node as input and materializes a set of its ancestor nodes as well as the

Algorithm 1: Rule to infer BGP RIB entry from main RIB entry.

```
1  def infer_from_main_rib_entry(f,
   ↪ stable_state):
2    if not (f is MainRibEntry and f.protocol ==
     ↪ 'bgp'):
3      return []
4    bgp_entry = stable_state.bgp_rib.lookup(
5      host=f.host,
6      prefix=f.prefix,
7      nexthop=f.nexthop,
8      status='BEST'
9    )
10   return [(bgp_entry, f)]
```

Algorithm 2: Rule to infer ancestors of a post-import BGP message.

```
1  def infer_from_bgp_message(m, stable_state):
2    if not (m is BgpMsg and m.is_post_import):
3      return []
4    bgp_edge = stable_state.bgp_edges.lookup(
5      recv_host=m.host
6      send_ip=m.nexthop
7    )
8    origin_entry = stable_state.bgp_rib.lookup(
9      host=bgp_edge.send_host,
10     prefix=r.prefix,
11     status='BEST'
12   )
13   pre_import_msg, export_clauses =
     ↪ policy_simulation(
14     input=origin_entry,
15     policy=bgp_edge.export_policy
16   )
17   _, import_clauses = policy_simulation(
18     input=pre_import_msg,
19     policy=bgp_edge.import_policy
20   )
21   return [(pre_import_msg, m), (bgp_edge, m)]
     ↪ +
22     [(cl, m) for cl in import_clauses] +
23     [(origin_entry, pre_import_msg), (bgp_edge,
       ↪ pre_import_msg)] +
24     [(cl, pre_import_msg) for cl in
       ↪ export_clauses]
```

edges the allows the ancestors to reach the input node. These nodes and edges will be merged into the materialized IFG by the construction algorithm. The implementation of these functions uses one or both of the *lookup-based inference* and *simulation-based inference*. Let us elaborate.

Lookup-based inference. The computation of data plane state is lossy. While a main RIB entry may be derived from a BGP RIB entry, we cannot infer the complete BGP RIB entry from the main RIB entry because BGP specific attributes (e.g., AS-path) are not preserved in the main RIB.

To handle this information loss, our inference takes two steps. It first infers attributes that can be known from heuristics (we know such heuristics from control plane semantic, *e.g.*, the BGP RIB entry should have the same prefix as the main RIB entry derived from it). Next, we look up all entries in the stable state that match the inferred attributes. For instance, Algorithm 1 shows the simplified function to infer the BGP RIB entry that led to a main RIB entry. Based on control plane semantics, if a main RIB entry indicates its source protocol to be BGP, it must have stemmed from a BGP RIB entry on the same router with the same `prefix` and `nexthop` attributes (Lines 5-7). Besides, the BGP RIB entry should have been selected as the best route (Line 8). Such information is enough to uniquely identify the parent within the known stable state. The return value (Line 10) is a list of tuples denoting the IFG edges materialized by this rule.

Simulation-based inference. Lookup-based inference falls short in two scenarios. First, when a parent fact is absent from the known stable state (e.g., routing messages), and second, when the heuristics fail to infer enough information so as to uniquely identify the parents (e.g., we cannot know which policy clauses are used in the production of a BGP route by looking at the resulted route). We use local simulations to complement lookup-based inference. But simulations can only be performed in the forward direction, *i.e.*, to compute a fact using simulations, we first need to know its parent. We use a generalized version of lookup-based inference to discover grandparent facts of a known fact, and then use simulations with the grandparents to infer their children (i.e., parents of the original fact).

Algorithm 2 shows the simplified inference rule that infers the ancestors of a post-import BGP message. Line 13 demonstrates the use of simulation-based forward inference to compute a missing parent fact on the fly. The two prerequisites to simulate the BGP message–the grandparent BGP RIB entry (`origin_entry`) and the BGP edge–are discovered via lookup-based backward inference, on Line 8 and Line 4 respectively. The simulation returns the derived BGP message after applying the routing policy, as well as the policy clauses exercised during the process. The second forward-simulation (Line 17) is to discover the policy clauses that are hit during the import process. The return value includes the inferred IFG edges that connect to the input node `m` as well as ones that connect to parent `pre_import_msg`. The former corresponds to information flow $m_i \leftarrow m_j, e_k, \{c_{l_1}, ...\}$ in Table 1 and the latter corresponds to $m_i \leftarrow r_j, e_k, \{c_{l_1}, ...\}$.

IFG construction. Next, we detail IFG materialization using inference rules. Assume for now that the information flow

Algorithm 3: IFG lazy materialization

Input: Initial nodes $\{v_i\}$; Inference rules $\{\phi_i : v \mapsto \{(u_i, v_i)\}\}$;
Output: Materialized IFG (V, E)
Data: Stable state data plane state (main RIB and protocol RIBs); Routing edges; Configuration elements;

```
1  Procedure BuildIFG({v_i}, {φ_i})
2      V, E ← {v_i}, ∅
3      V_prev ← {v_i}      // dirty nodes of previous iteration
4      while |V_prev| > 0 do
5          V_curr ← ∅      // dirty nodes of current iteration
6          foreach c ∈ V_prev do
7              foreach φ ∈ {φ_i} do
8                  E' ← φ(c)
9                  foreach (u_i, v_i) ∈ E' do
10                     if u_i ∉ V then
11                         V ← V ∪ {u_i}, V_curr ← V_curr ∪ {u_i}
12                     if v_i ∉ V then
13                         V ← V ∪ {v_i}, V_curr ← V_curr ∪ {v_i}
14                     if (u_i, v_i) ∉ E then E ← E ∪ {(u_i, v_i)}
15         V_prev ← V_curr
16     return (V, E)
```

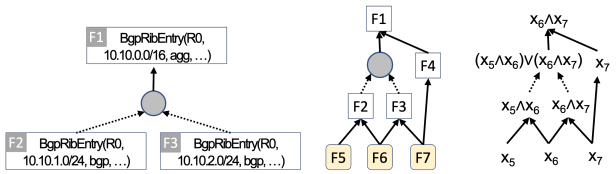
is deterministic; the next section discusses how we handle non-determinism.

As shown in Algorithm 3, the IFG initially contains only the nodes representing the tested data plane state facts from the input and does not have any edges (Line 2). It is then iteratively expanded by applying inference rules on existing nodes. In each iteration, all inference rules are applied to the dirty nodes derived from the previous iteration (Line 8). The new nodes and edges inferred during such process are collected and merged (with deduplication) into the IFG (Line 9-14). The computation repeats until no new facts can be derived in an iteration.

4.3 Handling uncertainty

There are situations where it is not certain which stable state facts contributes to a given fact. One such scenario is BGP aggregation, where a prefix (e.g., 10.10.0.0/16) is added to the RIB iff at least one more of its more specific prefixes (e.g., 10.10.1.0/24) is present. When multiple more specifics are present, we do not know which one triggered the aggregate. Another such scenario is when multiple paths are available for a routing edge to be established, which can happen when the network uses multipath routing. Here, we do not know which path is actually used by routing messages.

It is important to model and report such uncertainty because the notion of contribution is different. Unlike deterministic contribution, when the contribution is non-deterministic, one or more parent facts can disappear without impacting the outcome represented by the child. Our experiments have scenarios where 78% of the configuration lines have non-deterministic contribution, and the tested fact would not be



(a) (b) (c)

Figure 3: Modeling uncertainty. (a) BGP aggregate (F1) has two potential contributors. (b) F5 is weakly covered but F6 and F7 are strongly covered. (c) The predicates of IFG nodes.

impacted if any of them did not exist. Not separating such uncertain contribution would lead to misplaced confidence in how well configurations are tested.

We model contribution uncertainty using *disjunctive* nodes in the IFG. This node points to the parent fact (e.g., the aggregated RIB fact) and the multiple contributors to the parent point to this node. See Figure 3(a) for an example where a BGP aggregate could be triggered by either of the two more specific prefixes. When our inference rules encounter uncertainty during IFG materialization, they produce a disjunctive node and attach all contributors to it as children.

We introduce the notion of *weak* coverage to capture the configuration elements whose contribution to the tested facts is not critical. We define a contribution as non-critical if the tested fact will not be affected by deleting the configuration element from the IFG. In Figure 3(b), F5 is weakly covered when F1 is tested because F1 can be derived without any contribution from F5, via F2 and F6. On the other hand, F6 is strongly covered because, without it, neither F2 nor F3 can be derived and thus the disjunctive node cannot be derived. F7 is also strongly covered because it contributes to F4, which is essential to F1.

NetCov labels each covered configuration element as strong or weak after the materialization of the IFG. The label is determined as follows. We first assign a Boolean variable to each configuration element in the IFG. Next, we build a Boolean predicate of each IFG node on top of these variables. The predicate of a fact depends on the predicate of its ancestors in the IFG: A normal node depends on the conjunction of its immediate parents, and a disjunctive node depends on the disjunction of parents. Therefore the predicate of any IFG node is ultimately composed of the variables associated with configuration elements that lead to it, denoted as $\Gamma(v) = F(x_1, \ldots, x_n)$. Figure 3(c) shows the predicates of IFG nodes in Figure 3(b). We represent these Boolean predicates using Binary Decision Diagrams (BDDs) [8] and build BDD predicates by traversing the IFG. By definition, a configuration fact (denoted as x_i) is strongly covered if and only if there exists a tested data plane state fact (denoted as v), v is reachable from x_i in the IFG, and x_i is a necessary condition of $\Gamma(v)$. Therefore, once the predicates are built, we test graph reachability and logical necessity between each pair of configuration facts and

tested data plane facts. Necessity $\neg x_i \Rightarrow \neg\Gamma(v)$ is equivalent to unsatisfiability of $\neg x_i \wedge \Gamma(v)$. While (un)satisfiability is NP-Complete in general cases, we note that it is efficient in our case—it can be reduced to computing the cofactor $\Gamma(v)|_{x_i=0}$ and testing whether the cofactor is constant false, both of which are efficient using BDD operations.

We further reduce the size of BDD predicates by precluding configuration facts that can reach tested facts via a path with no disjunctive node, such as node F7 in Figure 3(b). These configuration facts must be strongly covered so their necessity do not need to be tested. Besides, their validity variables can be replaced with constant true when building BDD predicates, which will not affect the strong/weak classification of other configuration elements. We empirically find this heuristic to be effective in reducing the number of variables used for weak coverage computation.

4.4 Future Extensions

Our current model tracks the contribution of configuration elements to concrete data plane state entries. While this view aligns well with tools that perform data plane testing [50], data plane verification [25, 29], and control plane testing [12, 49], it is not applicable to control plane verification tools [1, 7] that reason about data plane symbolically (i.e., simultaneously reason about multiple data planes under different environments). Control plane verification tools turn configuration into an internal model that is used for validation. NetCov can be extended to these tools by tracking how configuration elements contribute to the model, akin to how compilers link program source information to its intermediate representations.

The current implementation of NetCov supports BGP, a path vector protocol, and static routes. Other protocols, including link state protocols (e.g., OSPF) and label switching protocols (e.g., MPLS) can be supported with appropriate extensions. Such extensions require defining protocol-specific configuration elements and data plane state facts (such as label information base entry for MPLS) as well as all new information flows.

5 Implementation

We implemented NetCov with 4,000 lines of Python code. A total of 18 lambdas (Python functions) encode the IFG inference rules. NetCov uses Batfish [6] to extract configuration elements from configuration files and to run targeted simulations, and it uses CUDD [37] for BDD operations.

NetCov supports several major router vendors supported by Batfish, including Arista, Cisco, and Juniper. It builds a vendor-neutral representation of configuration elements using vendor-specific information provided by Batfish. Table 2 lists the configuration elements that NetCov currently analyzes.

NetCov may not consider all components of a device's configuration. One category of such components is device management configuration (e.g., login settings), which does not impact data or control plane functionality. The second category is control plane components that are not currently modeled by NetCov. This includes IPv6 (which is not modeled by Batfish currently) and routing protocols other than BGP (e.g., OSPF). The presence of unconsidered components does not imply that NetCov cannot be used for that network. As we show in the next section, NetCov provides helpful coverage information for parts that are considered.

Type	Purpose
Interface	Interface and its settings (e.g., addresses)
BGP peer	BGP peer settings (e.g., IP address, AS number)
BGP peer group	BGP peer settings inherited by one or more peers
Route policy clause	One clause in an export or import route policy
Prefix list	List of prefixes, used in route policy clauses
Community list	List of BGP communities for route policy clauses
AS-path list	List of AS-path expressions for route policy clauses

Table 2: Configuration elements analyzed by NetCov.

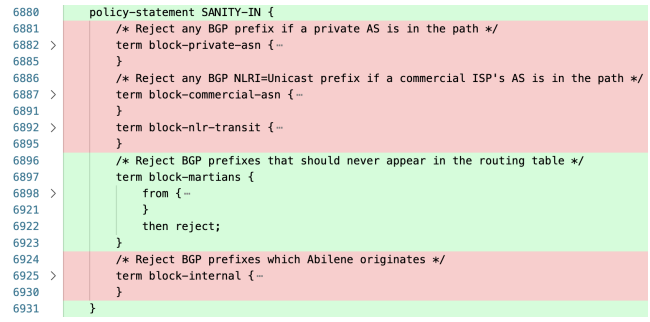
After constructing the IFG, which yields information on which configuration elements are covered, NetCov computes which lines are covered. NetCov leverages the Batfish parser to map configuration elements to line numbers. Each element typically spans multiple configuration lines, and when an element is covered, it deems all of those lines as covered.

Based on element and line coverage, NetCov produces three main outputs. The first is a coverage report at the granularity of individual lines (or elements). We produce this report in the `lcov` format, which is supported by common code coverage tools and enables users to visualize coverage results as annotations on configuration files. See Figure 4(a) for an example. The second is coverage aggregated at the file level, generated with the help of GNU LCOV [17]. See Figure 4(b) for an example. The third output is coverage aggregated by the type of configuration element, which shows what fraction of elements of each type are covered.

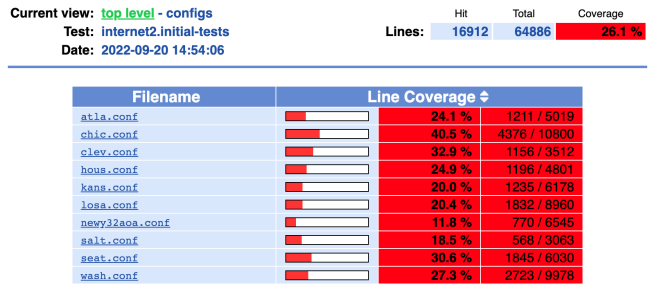
These outputs help users uncover testing gaps and improve their test suites in different ways. The aggregate results help identify systematic gaps such as "router A is poorly covered" or "routing policy clauses are poorly covered." The line-level results help them zoom in to specific gaps and develop tests that target them. The case study in the next section demonstrates this test suite improvement process.

6 Case Studies

We present case studies of using NetCov on two disparate networks, one a wide-area backbone and another a datacenter. In each case, using realistic test suites, we show that NetCov provides insight into what is and is not covered and how these insights help improve the test suites.



(a) Line-level coverage. Green background denotes covered lines, and red denotes uncovered lines. Some lines are collapsed for simplicity.



(b) File-level aggregate coverage. The overall coverage is at top right, and the coverage for individual files (devices) is in the table.

Figure 4: Example NetCov outputs.

6.1 Case Study I: The Internet2 backbone

Internet2 is a nation-wide network that connects over 60,000 US educational, research and government institutions. The routing design of Internet2 is typical of backbone networks. It has 10 BGP routers spread across the country. The routers are organized as a single autonomous system (AS), and they establish iBGP full mesh on top of internal reachability provided by the IS-IS protocol. The Internet2 routers connect to 279 external BGP peers, and heavily use route import and export policies. The import policy for an external peer has multiple policy statements, some specific to the peer and some shared within the same peer group. Peer-specific policies tend to specify a list of allowed prefixes from this peer, and others are used for sanity checking, preference setting, etc. Export policies are similarly structured.

Internet2's configurations that we study have 96,672 lines (in Juniper's JunOS format) across all routers. Of these, NetCov's coverage computation considers 64,886 lines. The bulk of the unconsidered lines correspond to device management, IPv6, and IS-IS protocol.

We do not have the data plane state of Internet2, which is needed to run data plane tests. We approximate it using Route Views [42], a repository of BGP routes from over two hundreds ASes worldwide. This data helps approximate BGP messages that external peers of Internet2 send to it. Consider a peer with AS number X. If we find a prefix P in RouteViews with AS-path $[A,X,Y]$, we assume that the peer sends P to Internet2 with AS-path $[X,Y]$. The existence of AS-path $[A,X,Y]$ means that AS A must have a route to P with AS-path $[X,Y]$, which it announces to its neighbors. If we find multiple AS-paths for a prefix, we pick the one with fewest AS hops.

We use these BGP messages that each peer sends to Internet2 as inputs to simulate Internet2's control plane using Batfish. The data plane state produced by this simulation is a coarse approximation of the real version, but it suffices to meet our goals of running data plane tests and characterizing configuration coverage.

6.1.1 Test suite coverage

To study how NetCov analyzes coverage for realistic test suites, we use the test suite proposed in Bagpipe [44]. It has three tests to validate Internet2's BGP configuration.

- *BlockToExternal*: ensure that BGP routes with BTE community are not announced to any external (eBGP) peer.
- *NoMartian*: ensure that incoming BGP messages from external peers for prefixes in the private address space ("Martian") are rejected.
- *RoutePreference*: ensure that if multiple routes to the same prefix are accepted from multiple external neighbors, the selected route belongs to the most preferred neighbor. The neighbor's preference depends on commercial relationship [13]. *Customers* are most preferred, followed by *p*eers, and then *providers*[4].

We implemented these tests using Batfish. *BlockToExternal* and *NoMartian* are control plane tests. *BlockToExternal* evaluates all BGP export policies on a set of BGP routes carrying the BTE community and asserts that the result be rejection. We generate the test cases by sampling BGP routes from the data plane state and attaching the BTE community to them. *NoMartian* evaluates all BGP import policies on a set of BGP routes destined for Martian addresses and asserts that the results be rejection. *RoutePreference* is a data plane test. It focuses on destination prefixes available via multiple neighbors and asserts that their local preferences reflect commercial relationship. We use CAIDA data [28] to infer commercial relationship between Internet2 and its BGP neighbors.

After running this test suite on Internet2, we find that it covers only 26.1% of configuration lines across all devices. Only a tiny fraction of configuration lines (0.5%) are weakly covered, so we do not separate weak/strong coverage for this case study; we will do that in the next one.

[4] As a not-for-profit network, Internet2 treats its member institutions as customers and other not-for-profit networks (such as ESNet) as peers. Internet2 does not have providers in its routing preference model.

To help understand what is and is not covered in more detail, NetCov enables network engineers to look at the data from multiple perspectives. Figure 4(b) shows per-device coverage. We see notable variation across devices, from 11.8% to 40.5%. As we show below, the test suite has systematic gaps, and the cross-device variation stems from different devices having different fractions of covered configuration elements.

Figure 5 shows the coverage broken down by the type of configuration elements. For simplicity, we create four buckets of element types, as shown in the legend. The bottom bar shows the fraction of reachable configuration lines in each bucket. The "Test Suite" bar shows the covered fraction of those lines, and the top three bars show the coverage of individual tests. The total coverage of individual tests is 0.6%, 0.9% and 24.7% respectively. *BlockToExternal* and *NoMartian* cover only one type of configuration element (routing policies), and even within this type, they cover a small fraction. *RoutePreference* covered all four buckets but its overall coverage is still limited.

Finally, NetCov reports that 27.9% of configuration lines are "dead code" that will never be exercised. They include defined BGP peer groups with no members and defined routing policies that are never used for any peer.[5]

With 69% of BGP configurations, 85% of interfaces, 88% of routing policies, and 57% of route attribute match lists being completely untested, this test suite is clearly undertesting the network. This leaves the network vulnerable to bugs in untested configurations elements. Prior to NetCov, it was not possible for network engineers to get any insight into the quality of their test suite. It was also not possible for them to get help toward systematically improving tests. We demonstrate this test suite improvement process next.

6.1.2 Coverage-guided test development

NetCov's feedback enables a test suite development process that enables users to systematically improve coverage, which helps test more critical aspects of the network and prevent outages. This process is iterative. In each iteration the user first identifies specific testing gaps and then creates new tests to target those gaps. We demonstrate the process using three iterations that focus on different types of gaps.

Iteration 1. We saw that routing policy coverage of *NoMartian* test is low (Figure 5) despite that it checks the import policies for all external peers. To investigate, we look at the structure of Internet2 import policies and find that routers have a policy named `SANITY-IN` which is shared by the majority of external neighbors. Figure 4(a) shows this policy with annotated coverage. Each router has an independent copy of this policy, but the copies and the coverage results are identical across routers. Of the five clauses in the policy, the clause `block-martians` starting at line 6,896 is the only clause that is covered. This coverage result confirms that the *NoMartian* test did its job, and more importantly, it revealed a systematic testing gap–the other four classes of forbidden routes are not being tested.

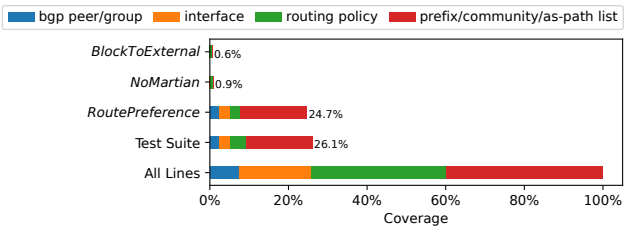


Figure 5: Coverage of the initial test suite broken down to each individual test and configuration type.

Once we know the gap, the solution suggests itself. We added a new test, *SanityIn*, to enforce that the other four classes of received BGP messages should be rejected. After adding this test, we used NetCov to confirm that this testing gap had been addressed. Routing policy coverage was improved by 0.6% and all five terms of `SANITY-IN` were covered by the new test suite. The quantitative improvement is low because `SANITY-IN` is just one of many policies in the network. With feedback from NetCov, network engineers can identify testing gaps in other routing policies and add more tests in a similar way.[6]

Iteration 2. BGP peer configuration coverage of *RoutePreference* test in Figure 5 is surprisingly low, given that all external BGP peers are supposed to be checked. Upon further investigation we find that the uncovered peers have permitted prefix-lists that do not overlap with other peers' lists, which left these peers untested.

We added a new test, *PeerSpecificRoute*, to check that BGP announcements received from external peers should be accepted if their prefixes is in a peer-specific prefix list. This test improved BGP peer coverage from 32% to 46%. The rest of untested BGP peers are either not allowed to send BGP routes to Internet2 or is intended for other internal use, such as monitoring and management. This test also improved prefix-list coverage from 45% to 63%. The remaining of untested prefix-lists are mostly (30% out of 37%) ones that are defined by never referenced.

Iteration 3. The low coverage of interface configuration in Figure 5 reveals another testing gap. *RoutePreference* is the only test in the initial test suite that checks interface configurations, and it only considers one category of interfaces–ones that are used to establish the tested BGP edges. Many other

[5] Per best practices, these lines should be deleted. Or, at a minimum, they should be tested lest someone start using an unused, erroneous policy. When it comes to testing, such lines can never be exercised by data plane tests, though control plane tests may be written for them.

[6] Automatic test generation based on coverage feedback will further help engineers. We will investigate this in the future.

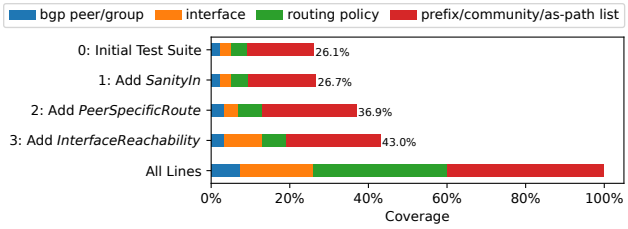


Figure 6: Coverage improvement with test suite iterations.

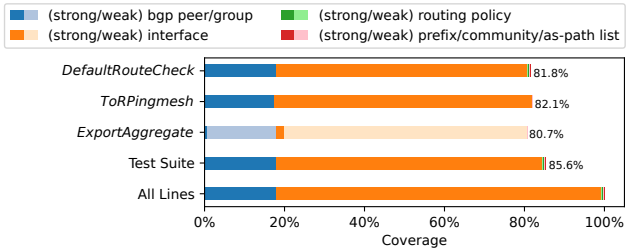


Figure 7: Coverage of synthetic datacenter network for different tests and types of configuration elements.

interfaces remain untested, including but not limited to ones that associate with untested BGP edges and other routing protocols, and the ones that are unused.

We added a new PingMesh-style [18] test, *InterfaceReachablility*, to check that the IPv4 addresses assigned to interfaces should be reachable from each router in the network. This test increased interface coverage from 15% to 53%. The rest of untested interfaces do not have IPv4 addresses assigned.

Figure 6 summarizes the coverage improvement for the three iterations of test improvement in our study. After only three iterations, the overall coverage was improved from 26% to 43%. This final coverage number is far from perfect, but our goal was not to develop the ideal test suite for Internet2; we wanted to demonstrate how coverage information helps develop new tests. Networks are complex, and we should not expect to get the job done with 6 tests. Many more tests are likely needed. With NetCov, network engineers now have a tool to develop new tests that meaningfully improve coverage.

6.2 Case study II: Datacenter networks

We study the coverage for data center networks which have a different topology and routing design. We create synthetic fat-tree [2] networks with routers across three tiers. The leaf routers at the bottom tier connect to hosts. Aggregation routers at the middle tier connect to leaf routers in a pod and to spine routers at the top tier. The spine routers connect to the wide area network (WAN). The WAN is not part of the tested network. Each leaf router is assigned a /24 prefix which is advertised inside the data center through eBGP. Spine routers receive a default route (prefix 0.0.0.0/0) from WAN via eBGP and propagate it to lower tiers. At each spine router, the entire address space of the network is summarized into a /8 prefix and is announced to WAN. Multipath routing (ECMP) is enabled with maximum number of paths set to 4. Routing policies are only configured at spine routers to white-list the default route received from WAN peers. We synthesize the configurations of these networks in Cisco IOS format.

We study a test suite of three tests inspired in prior works on data center network validation [18, 23].

- *DefaultRouteCheck*: ensure that each router has the default route.
- *ToRPingmesh*: ensure that each leaf router's assigned subnet is reachable from all other leaf routers.
- *ExportAggregate*: ensure that each spine router exports the aggregate route to WAN.

Figure 7 shows the coverage result when the network has a total of 80 routers. Given the uniformity of the network and the test suite, coverage results are similar for other network sizes. The total coverage of individual tests is 81.5%, 82.1% and 80.7% respectively, and the three tests together cover 85.3% of configuration lines. We find that these tests cover largely the same configuration elements—interfaces and BGP peerings between the data center routers—despite checking for seemingly different network behaviors. This result indicates that test development without coverage feedback can be ineffective in terms of covering the testing gaps.

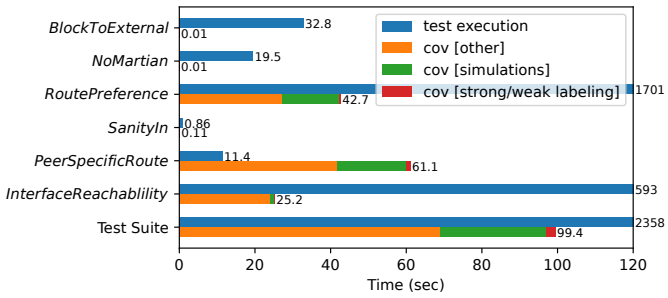
The coverage of *ExportAggregate* shows a large proportion of weak coverage. This is because a spine router has routes to all leaf routers, so that all leaf subnets contribute to the tested aggregate route, albeit weakly. Separating out weak coverage here avoids false negatives of testing gaps—the aggregate routes would be there even if some of the BGP peering or interfaces are misconfigured, therefore testing the aggregate routes provides a weaker endorsement for the covered BGP peerings and interfaces to be bug-free.

By looking at uncovered configuration lines reported by NetCov, we learn that most correspond to host-facing interfaces on leaf routers. Adding tests that target those interfaces improves this test suite and eliminate testing gaps. We omit results of this iteration.

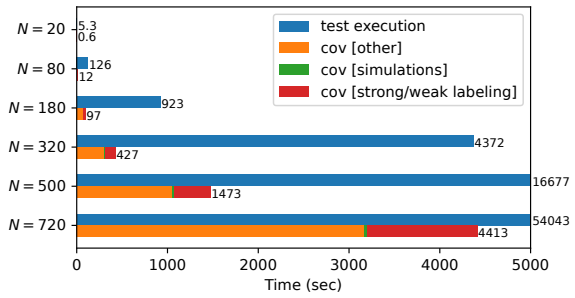
7 Performance Evaluation

We benchmark the performance of NetCov on both types of networks we studied above. Our test machine has two Intel Xeon CPUs (16 core each, 3.1 Ghz), 384 GiB of DRAM, and runs Ubuntu 18.04.

Figure 8(a) shows the time to compute coverage for each test in §6.1 and for the full test suite. It breaks out the time spent on simulations and strong/weak labeling, and, for reference, also shows the test execution time. We see that coverage



(a) Internet2.



(b) Fat-tree networks.

Figure 8: Time to compute coverage.

computation is reasonably fast. The full test suite takes only 99.4 seconds. In comparison, the test execution takes 2,358 seconds. The total coverage computation time is less than the sum for individual tests because facts tested by multiple tests are tracked only once. The graph also shows that simulations and strong/weak labeling are a minority component, which means that most of the time is spent on walking the IFG and doing lookups in stable state for backward inference.

Figure 8(b) shows test execution and coverage computation time for the test suite in §6.2, as a function of the data center network size. Coverage computation takes 4,413 *sec* on the largest network, which has 2,040,624 RIB entries. This time is less than 9% of the time to execute the test suite. While substantial, we deem it acceptable in practice. Configuration coverage analysis can be run in the background, as code coverage is often run. NetCov does not slow down test execution, which is on the critical path to finding configuration errors and updating the network.

However, time to compute coverage increases rapidly with network size. This is because the number of RIB entries grows quadratically and so does the number of vertices in the IFG. We find that the average time to materialize an IFG node does not change substantially because all computation is local to the node. The scaling trends suggest that to scale NetCov to much larger networks, we need a concurrent implementation of IFG materialization. Our current implementation is single-threaded (as Python interpreter is single-threaded).

8 Comparison to Data Plane Coverage

We demonstrate the unique value of control plane coverage by comparing it to data plane coverage. Following Yardstick [47], we quantify data plane coverage as the proportion of main RIB (forwarding) rules exercised. Figure 9 shows the comparison for different cases. Figure 9(a) shows the comparison for Internet2 for all tests in §6.1 and a hypothetical data plane test that inspects all main RIB rules. Figure 9(b) shows the comparison for fat-tree tests in §6.2.

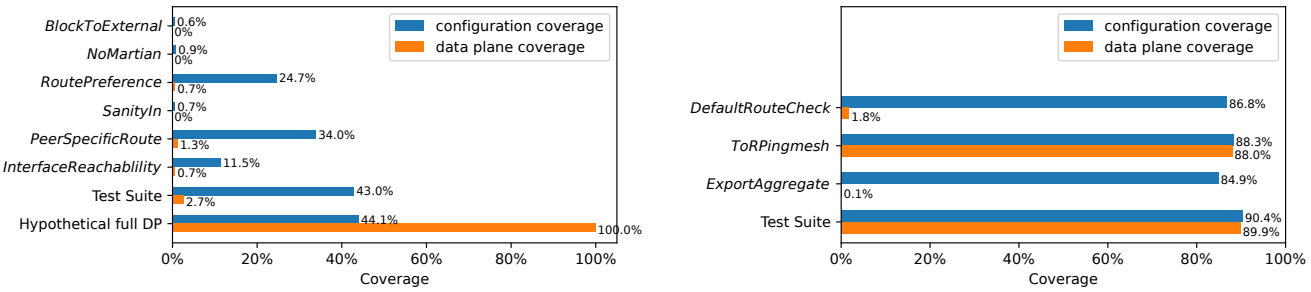
Besides the obvious advantage that only control plane coverage can support control plane tests—the graphs show 0% data plane coverage for these tests—there are two main advantages to using control plane coverage to guide network test development. First, it reveals testing gaps that can not be revealed by data plane coverage. Tests with high data plane coverage do not necessarily have high control plane coverage, as we can see in the last row of Figure 9(a). Covering 100% of the data plane state covered only 41% of the configuration. If the engineers were to improve the test quality under the guidance of only data plane coverage, they would not know that 59% of the configurations remain untested. The reason of this disagreement is that some configuration lines are only exercised under specific environments (failures, routing messages). For instance, list-filtered route policies apply on BGP messages within a specific range, and will only be exercised when such messages appear in the environment.

Second, testing more data plane state can sometimes be redundant in covering configurations, when the tests hit the same configuration elements. For example, the *DefaultRouteCheck* test in Figure 9(b) has only 1.8% data plane coverage because it only tests default routes, which is a small fraction of all main RIB routes. However, because correct propagation of default routes incorporates many BGP peerings and interfaces in the network, this test has extensive configuration coverage (87%). The *ToRPingmesh* test covers much more data plane state (88%), but adding it atop *DefaultRouteCheck* has little value because this state is derived from almost the same set of configurations lines. We do not necessarily imply that engineers should drop one of these tests, as there may be other reasons to keep both. Our observations are about their value toward configuration coverage.

9 Related Work

Our work builds on top of four lines of research.

Code coverage. We borrow from the software domain the idea of using code coverage to reveal testing gaps, quantify test suite quality, and help engineers improve their test suites [4, 15, 20]. Our coverage analysis techniques, however, are specialized to the operation of network configurations.



(a) Internet2.

(b) Fat-tree with k=10.

Figure 9: Comparing control plane and data plane coverage.

Data plane coverage. Yardstick introduced data plane coverage metrics [47] that quantify the proportion of data plane elements such as forwarding rules and paths that are exercised by network tests. Configuration coverage goes further and maps tested data plane components to configuration elements that contribute to them. It provides more direct feedback because network engineers author configurations, not data plane state, and it supports testing of configuration elements that are not exercised by the current data plane state.

Network testing and verification. A range of tools can analyse properties of network data and control planes [7, 12, 14, 18, 19, 23, 25, 26, 48, 49]. NetCov borrows ideas from verification tools to concisely model the network, *e.g.*, focusing on stable state and routing protocol instances [7, 14]. However, NetCov target a different problem—reveal what is tested vs enabling testing of new properties–and uses different techniques.

Network provenance. Provenance systems can track causal dependencies of events in distributed systems. Provenance systems like ExSPAN [52] materialize provenance graphs by tracing system execution in forward direction. Negative provenance systems can reason about missing events [46] and materialize provenance graphs lazily using backward inference. NetCov too uses a graph-based model. However, it is unique in terms of accommodating network configuration into a provenance model, and this model, tailored to the stable state assumption, is more succinct. Further, it combines backward and forward inference to overcome the limitations of using only one type of inference.

Software configuration testing. As for networks, configuration testing is an important problem for software systems as well. Sun et al. developed a system that can link software tests to exercised configuration parameters [38]. They exploit dependence on configuration settings being explicit, observable via read/write operations that use standard get/set APIs. NetCov targets a setting where the dependencies are implicit and non-local. Routers read the entire configuration file, and their forwarding behavior depends on that file and information received from neighbors who in turn act based on their configuration files and their neighbors. That led us to develop a different approach to tracking configuration dependencies. We will investigate in the future if our approach can be extended to software systems where dependence between tested runtime behavior and configuration is not explicit.

10 Summary

NetCov reveals which configuration lines are tested by a suite of network tests. It uses an information flow model based on control plane semantics to track which configuration lines contribute to tested data plane state. It accounts for non-local and non-deterministic contributions, and for performance, it discovers the graph lazily. Our experiments showed that NetCov successfully reveals coverage gaps for real-world networks and test suites, and these tests can have surprisingly low coverage, e.g., 26% of configuration lines for Internet2. They also showed how its feedback helps improve coverage.

Acknowledgments

We thank the NSDI'23 reviewers and our shepherd, Aditya Akella, for feedback on the earlier version of this paper. This work was supported in part by NSF grant CNS-2007073 and Cisco Systems.

Ethical considerations

This work does not raise any ethical issues.

References

[1] Anubhavnidhi Abhashkumar, Aaron Gember-Jacobson, and Aditya Akella. Tiramisu: Fast multilayer network verification. In *Proceedings of NSDI 20*, pages 201–219. USENIX Association, 2020.

[2] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network

architecture. In *Proceedings of SIGCOMM '08*, page 63–74. ACM, 2008.

[3] Mae Anderson. Time Warner cable says outages largely resolved. http://www.seattletimes.com/business/time-warner-cable-says-outages-largely-resolved, 2014.

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

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

[6] Batfish: Network configuration analysis tool. https://github.com/batfish/batfish.

[7] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. A general approach to network configuration verification. In *Proceedings of SIGCOMM '17*, pages 155–168. ACM, 2017.

[8] Karl S Brace, Richard L Rudell, and Randal E Bryant. Efficient implementation of a BDD package. In *Proceedings of the 27th ACM/IEEE design automation conference*, pages 40–45, 1991.

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

[10] Cisco Systems, Inc. Configure protocol redistribution for routers. https://www.cisco.com/c/en/us/support/docs/ip/enhanced-interior-gateway-routing-protocol-eigrp/8606-redist.html.

[11] Codecov. Codecov: The leading code coverage solution. https://about.codecov.io/, 2021.

[12] Ari Fogel, Stanley Fung, Luis Pedrosa, Meg Walraed-Sullivan, Ramesh Govindan, Ratul Mahajan, and Todd Millstein. A general approach to network configuration analysis. In *Proceedings of NSDI 15*, pages 469–483. USENIX Association, 2015.

[13] Lixin Gao and Jennifer Rexford. Stable internet routing without global coordination. *IEEE/ACM Transactions on networking*, 9(6):681–692, 2001.

[14] Aaron Gember-Jacobson, Raajay Viswanathan, Aditya Akella, and Ratul Mahajan. Fast control plane analysis using an abstract representation. In *Proceedings of SIGCOMM '16*, pages 300–313. ACM, 2016.

[15] Milos Gligoric, Alex Groce, Chaoqiang Zhang, Rohan Sharma, Mohammad Amin Alipour, and Darko Marinov. Comparing non-adequate test suites using coverage criteria. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ISSTA 2013, page 302–313, 2013.

[16] Timothy G Griffin, F Bruce Shepherd, and Gordon Wilfong. The stable paths problem and interdomain routing. *IEEE/ACM Transactions On Networking*, 10(2):232–243, 2002.

[17] GNU Guix. lcov–code coverage tool that enhances gnu gcov. https://guix.gnu.org/en/packages/lcov-1.15/.

[18] 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. Pingmesh: A large-scale system for data center network latency measurement and analysis. In *Proceedings of SIGCOMM '15*, page 139–152. ACM, 2015.

[19] Alex Horn, Ali Kheradmand, and Mukul Prasad. Delta-net: Real-time network verification using atoms. In *Proceedings of NSDI 17*, pages 735–749. USENIX Association, 2017.

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

[21] Istio. Diagnose your configuration with istioctl analyze. https://istio.io/latest/docs/ops/diagnostic-tools/istioctl-analyze/.

[22] Marko Ivanković, Goran Petrović, René Just, and Gordon Fraser. 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*, pages 955–963. ACM, 2019.

[23] 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. Validating datacenters at scale. In *Proceedings of SIGCOMM '19*, pages 200–213. ACM, 2019.

[24] Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. *IEEE transactions on software engineering*, 37(5):649–678, 2010.

[25] Peyman Kazemian, George Varghese, and Nick McKeown. Header space analysis: Static checking for networks. In *Proceedings of NSDI 12*, pages 113–126. USENIX Association, 2012.

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

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

[28] Matthew Luckie, Bradley Huffaker, Amogh Dhamdhere, Vasileios Giotsas, and KC Claffy. AS relationships, customer cones, and validation. In *Proceedings of IMC '13*, pages 243–256, 2013.

[29] Haohui Mai, Ahmed Khurshid, Rachit Agarwal, Matthew Caesar, P. Brighten Godfrey, and Samuel Talmadge King. Debugging the data plane with Anteater. In *Proceedings of SIGCOMM '11*, pages 290–301. ACM, 2011.

[30] Netcov: Network configuration coverage tool. https://github.com/UWNetworksLab/netcov.

[31] Santhosh Prabhu, Kuan Yen Chou, Ali Kheradmand, Brighten Godfrey, and Matthew Caesar. Plankton: Scalable network configuration verification through model checking. In *Proceedings of NSDI 20*, pages 953–967. USENIX Association, 2020.

[32] Bruno Quoitin and Steve Uhlig. Modeling the routing of an autonomous system with C-BGP. *IEEE network*, 19(6):12–19, 2005.

[33] Steve 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.

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

[35] Deon Roberts. Facebook says its outage was caused by a cascade of errors. https://www.nytimes.com/2021/10/05/technology/facebook-outage-cause.html, 2021.

[36] Joao Luis Sobrinho. Network routing with path vector protocols: Theory and applications. In *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 49–60, 2003.

[37] Fabio Somenzi. CUDD: CU decision diagram package release 2.5.0. *University of Colorado at Boulder*, 2012.

[38] Xudong Sun, Runxiang Cheng, Jianyan Chen, Elaine Ang, Owolabi Legunsen, and Tianyin Xu. Testing configuration changes in context to prevent production failures. In *Proceedings of OSDI'20*. USENIX Association, 2020.

[39] Yevgeniy 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.

[40] 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. Safely and automatically updating in-network ACL configurations with intent language. In *Proceedings of SIGCOMM '19*, page 214–226. ACM, 2019.

[41] Omer Tripp, Marco Pistoia, Stephen J Fink, Manu Sridharan, and Omri Weisman. Taj: effective taint analysis of web applications. *ACM Sigplan Notices*, 44(6):87–97, 2009.

[42] Route Views. University of Oregon Route Views project. http://www.routeviews.org/routeviews/, 1997.

[43] Rosemary Wang. Testing HashiCorp Terraform. https://www.hashicorp.com/blog/testing-hashicorp-terraform.

[44] Konstantin Weitz, Doug Woos, Emina Torlak, Michael D Ernst, Arvind Krishnamurthy, and Zachary Tatlock. Scalable verification of border gateway protocol configurations with an SMT solver. In *Proceedings of OOPSLA 2016*, pages 765–780. ACM, 2016.

[45] Zach Whittaker. T-mobile hit by phone calling, text message outage. https://techcrunch.com/2020/06/15/t-mobile-calling-outage/, 2020.

[46] Yang Wu, Mingchen Zhao, Andreas Haeberlen, Wenchao Zhou, and Boon Thau Loo. Diagnosing missing events in distributed systems with negative provenance. *ACM SIGCOMM Computer Communication Review*, 44(4):383–394, 2014.

[47] Xieyang Xu, Ryan Beckett, Karthick Jayaraman, Ratul Mahajan, and David Walker. Test coverage metrics for the network. In *Proceedings of SIGCOMM '21*, page 775–787. ACM, 2021.

[48] Hongkun Yang and Simon S. Lam. Real-time verification of network properties using atomic predicates. *IEEE/ACM Trans. Netw.*, 24(2):887–900, April 2016.

[49] 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. Accuracy, scalability, coverage: A practical configuration verifier on a global WAN. In *Proceedings of SIGCOMM '20*, page 599–614. ACM, 2020.

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

[51] Hongyi Zeng, Shidong Zhang, Fei Ye, Vimalkumar Jeyakumar, Mickey Ju, Junda Liu, Nick McKeown, and Amin Vahdat. Libra: Divide and conquer to verify forwarding tables in huge networks. In *Proceedings of NSDI 14*, pages 87–99. USENIX Association, 2014.

[52] Wenchao Zhou, Micah Sherr, Tao Tao, Xiaozhou Li, Boon Thau Loo, and Yun Mao. Efficient querying and maintenance of network provenance at internet-scale. In *Proceedings of SIGMOD '10*, pages 615–626. ACM, 2010.