Experience Track

Lessons from the evolution of the Batfish configuration analysis tool

Matt Brown
Intentionet

Ari Fogel
Intentionet

Daniel Halperin
Intentionet

Victor Heorhiadi
Intentionet

Ratul Mahajan
Intentionet
University of Washington

Todd Millstein
Intentionet
UCLA

ABSTRACT

Batfish is a tool to analyze network configurations and forwarding. It has evolved from a research prototype to an industrial-strength product, guided by scalability, fidelity, and usability challenges encountered when analyzing complex, real-world networks. We share key lessons from this evolution, including how Datalog had significant limitations when generating and analyzing forwarding state and how binary decision diagrams (BDDs) proved highly versatile. We also describe our new techniques for addressing real-world challenges, which increase Batfish performance by three orders of magnitude and enable high-fidelity analysis of networks with thousands of nodes within minutes.

CCS CONCEPTS

• **General and reference** → *Verification*; • **Networks** → **Network manageability**.

KEYWORDS

Network verification, configuration analysis, Batfish

ACM Reference Format:
Matt Brown, Ari Fogel, Daniel Halperin, Victor Heorhiadi, Ratul Mahajan, and Todd Millstein. 2023. Lessons from the evolution of the Batfish configuration analysis tool. In *ACM SIGCOMM 2023 Conference (ACM SIGCOMM '23), September 10–14, 2023, New York, NY, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3603269.3604866

1 INTRODUCTION

Batfish is a tool to validate network configurations before they are applied to the network. It uses a model of the network control plane (e.g., routing protocols like BGP) to derive the data plane that will result from the configurations, and then it verifies that the data plane adheres to the network's availability and security policies (e.g., ensure that no packet from A reaches B).

Batfish is in use at many large enterprises [12, 30, 31, 46, 53, 59]. It also serves as the backend for the Oracle Network Path Analyzer [48], which analyzes cloud virtual networks. Batfish has a large user community of network engineers, with over 1700 members in Slack channels [54, 55], and many research tools are built on top of it [1, 6, 7, 13, 19–21, 29, 37, 58, 60]. Batfish began as a research project a decade ago [16]. It has evolved significantly since then into an industrial-strength tool. Since its original development, we added support for many new router vendors (e.g., Arista, SONiC), stateful firewalls (e.g., Palo Alto, Check Point), load balancers (e.g., F5, A10), advanced network protocols (e.g, EVPN, VXLAN), and virtual cloud networks (e.g., AWS). We re-architected key parts of Batfish to overcome challenges related to performance and usability that we faced along the way.

This paper describes five lessons we learned from Batfish's evolution. We hope that these lessons, also relevant for other network modeling and analysis tools, will inform the design and development of the next generation of tools.

Lesson 1: Datalog was great for prototyping, but not for production use. The original version of Batfish used Datalog, a declarative logic programming language, to model the network control plane, derive the data plane, and run verification queries. We removed all uses of Datalog because it was difficult to express complex control plane behaviors and we lacked control over execution order, which is key to performance and deterministic convergence.

Lesson 2: Binary decision diagrams (BDDs) are great for data plane analysis. There is a longstanding tradeoff for data plane verification between the ease of tool development and performance [62]. Tools based on general solvers (e.g., SAT/SMT, Datalog) [43, 45, 56, 57] are easier to develop because much heavy lifting can be outsourced, but they face performance challenges because domain-specific optimizations can be hard to express. Custom encodings [33, 61, 62], on the other hand, perform well but are hard to build and extend. We discovered that casting data plane analysis as a dataflow analysis [35] using BDDs [2] can resolve this tradeoff and provide the best of both worlds. It let us compactly encode rich data plane behaviors and express many optimizations.

Lesson 3: Analysis fidelity is hard, but not why you think. Batfish uses a model of how the router software interprets the configuration instead of running actual software. The risk that the model might deviate from the actual router behavior is commonly attributed to bugs or evolution of router software; after a minor version change, Cisco IOS may interpret some configuration commands differently. Instead, we found that most modeling issues stem from the diversity of language features and undocumented interactions among them.

All authors are currently also with Amazon Web Services. The work reported in this paper was done while they were with Intentionet and other listed institutions.

ACM SIGCOMM '23, September 10–14, 2023, New York, NY, USA

ACM ISBN 979-8-4007-0236-5/23/09.
https://doi.org/10.1145/3603269.3604866

Lesson 4: Usability is hard for reasons you think, and then some. Batfish faces usability challenges seen in other verification domains [9, 51], such as the lack of precise specifications. However, it also faces challenges unique to networking. Network engineers are familiar with concrete testing tools like `traceroute`, which take a specific flow (combination of starting location and packet header) as input and output specific paths of that flow. Using network verification tools requires a mental shift, where engineers must reason about large sets of flows and paths as input and output. The completeness of analysis can also uncover many scenarios that technically violate tested properties but are uninteresting to users (e.g., when guaranteeing reachability from A to B for *all* traffic, traffic with spoofed source IPs may be uninteresting). A haystack of such uninteresting violations can easily conceal real issues.

Lesson 5: Deep configuration modeling has many applications. Though the original goal of Batfish was analysis of network forwarding, we discovered that network engineers wanted the tool to check many other properties, such as the compatibility of BGP configuration across neighbors and the uniqueness of assigned IP addresses in the network. Fortunately, we found that Batfish's detailed model of network configurations and their behavior—a prerequisite for data-plane generation—made it easy to support these other applications.

The rest of this paper elaborates on these lessons and describes how the design of Batfish evolved. It includes our current approach to configuration modeling and generating the data plane in a scalable, deterministic manner; the new BDD-based engine for data plane verification; and the techniques we developed to improve usability and analysis fidelity.

We also describe how network engineers use Batfish today and benchmark the performance of the current version using data from 11 real networks. Compared to the original version, the new techniques improve data plane generation by three orders of magnitude and data plane verification by an order of magnitude. The analysis finishes in minutes, even on networks with thousands of nodes.

These results and the large user community confirm that Batfish has evolved and matured considerably, but challenges remain. We conclude the paper by discussing the main ones, including the lack of automation inside most networks, modeling diverse forwarding pipelines, and current bottlenecks for usability and scaling.

2 ORIGINAL BATFISH

Batfish was originally developed to analyze network forwarding proactively, i.e., before configuration changes are deployed. This capability helps network engineers find configuration errors that cause undesirable packet forwarding—when a packet that should reach its destination is dropped, or vice versa—before the network is impacted by the error.

Before Batfish, there were two main approaches to finding network configuration errors. The first statically analyzed configuration text [3, 14, 47, 64] to proactively find a class of errors (e.g., a BGP session is not configured on both ends). However, this approach did not model the semantics of configurations in sufficient detail and could not analyze packet forwarding. The second approach analyzed snapshots of the network's data plane state (i.e., forwarding tables) pulled from the live network [33, 45]. Though it could analyze packet forwarding , it could not find configuration errors proactively because it needed to pull state from the network. Batfish aimed to provide the best of these two approaches.

The original Batfish used a *model-based approach*: a high-fidelity model simulated the network control plane based on its topology and configurations to produce the resulting data plane state. It then verified that this state matched engineers' expectations. We chose Datalog for control plane modeling because it let us outsource to a Datalog engine tasks such as simulation and tracking provenance of violations. Batfish had a four-stage workflow.

Stage 1: Configuration parsing and modeling. The first stage translated configuration text of all network routers to a normalized representation. Unlike the configuration text, where the syntax is specific to a router OS (e.g., Arista EOS, Cisco IOS), the internal representation was vendor-independent. Specifically, a configuration was represented as a collection of logical *facts* in a variant of Datalog used by the LogicBlox engine [40]. If the configuration of node `N` declared an OSPF link cost of 500 on interface `I`, then we produced the Datalog fact `OspfCost(N, I, 500)`, where `OspfCost` is a Datalog predicate that we defined. There were similar predicates for all other aspects of network configurations that affect routing, and we populated facts about these predicates.

This stage used Antlr [50] to parse the configuration text into an *abstract syntax tree* (AST) and then converted the AST to Datalog facts. This parsing-based approach was a departure from prior configuration analysis tools [14], which extracted a few specific configuration elements of interest using regexes. It was motivated by the need to extract all configuration elements that could impact the data plane state.

Stage 2: Data plane state computation. The second stage derived the data plane state from the configuration facts and a user-provided *environment*. The environment, also represented internally as Datalog facts, included link states (up/down) and routing messages from external neighbors. Batfish defined a model of the network control plane, including protocols like BGP and OSPF, static routes, and route redistribution. The model was essentially a function that took the two preceding inputs and produced the data plane state of each node.

The model was encoded as Datalog *rules* that specified how to derive new facts from existing facts. Rules defined how to compute OSPF routes, encoded as facts like `OspfRoute(node, nextHop, nextHopIP, cost)`, from other facts, such as the OSPF link cost fact. These rules were recursive since the OSPF routes at one node depend on those at its neighbors; Datalog's support for recursion made it a natural fit. There were similar rules for other protocols, additional rules that determined best routes across protocol-specific best routes, and finally rules that produced the forwarding state as facts of the form `Forward(node, flow, neighbor)` and `Drop(node, flow)`. Batfish used the LogicBlox Datalog engine [40] to derive all facts implied by these rules and the initial set of facts until reaching a fixed point, resulting in a representation of the network's data plane state.

Stage 3: Data plane verification. The third stage verified the network's forwarding behavior. Its inputs were the data plane state computed above and a property of interest (e.g., HTTP packets should be able to reach B from A), encoded in first-order logic.

The output was either a "thumbs up" (the property holds) or a counterexample with a concrete packet header and starting node.

This stage used Network Optimized Datalog (NoD) [43] and the Z3 SMT solver. NoD took as inputs the data plane state and the (negation of the) property and, if the property were violated, output a boolean formula for constraints on the set of packets that violated the property. Batfish then used Z3 to derive a concrete counterexample from the constraints.

Stage 4: Explaining violations. This final stage aimed to help users understand why their property was violated. It translated the concrete counterexample packet into the Datalog model of the data plane and used LogicBlox to simulate the behavior of this flow through the network. This simulation produced a set of Datalog facts that identified all forwarding rules the counterexample packet touched along its path(s).

The produced facts included information on the specific route used at a node, how the node learned that route (e.g., via OSPF), and how that route was produced from the configuration. Producing this extra information was trivial in Datalog since the dependencies among facts and rules are explicit.

3 LESSONS FROM EVOLUTION

Faced with analyzing large real-world networks, the design and implementation of Batfish evolved significantly. While the model-based approach and the four-stage workflow remained, we completely re-wrote Stages 2, 3, and 4, sometimes multiple times. We also significantly changed Stage 1 and created an additional, offline stage focused on analysis fidelity. We take a retrospective view and first highlight five key lessons from this evolution. In the next section, we describe the new Batfish components.

Lesson 1: Datalog was great for prototyping but not for production use

The original Batfish used Datalog extensively, which enabled use of an off-the-shelf solver and simplified development. However, early in the journey of transforming Batfish to a real-world tool, the limitations of a Datalog-based control-plane model became critical roadblocks. We encountered three main limitations, which are relevant not only for network modeling but also for network protocol implementations [41, 42].

1. Expressiveness. As we faced more complex configuration constructs, it became increasingly difficult to precisely model semantics in Datalog. For example, route maps can use regular expressions and arithmetic. Datalog is not Turing-complete (though LogicBlox's Datalog variant had some extensions), so it may be impossible to exactly model semantics in some cases. Even when we found a way to encode certain advanced semantics, it often required a subtle, unwieldy encoding that was hard to understand and maintain.

2. Performance. For large networks, Batfish's data plane generation was too slow. With a Datalog solver, we did not control the order in which rules and facts were considered. Indeed, not having to specify execution order is a big advantage of declarative models. However, ordering has a huge impact on performance for routing computation. BGP best paths are based on IGP (e.g., OSPF) best paths. It is thus better to first compute IGP paths and then compute BGP paths. A Datalog solver might instead compute BGP paths based on the current IGP paths, which then required the BGP paths to be updated all over again after IGP paths were updated.

Additionally, the Datalog engine retains in memory all intermediate facts, including routes that are eventually deemed sub-optimal and discarded. With non-monotonic routing computations (as is the case here), with no additional context, this retention is necessary [44]. However, it also consumes substantial memory. Many intermediate facts are unnecessary to retain, but in a declarative model we lacked a way to specify which intermediate facts should be retained and which could be safely forgotten.

3. Deterministic convergence. We encountered networks where, unless execution order is controlled (which is not possible with Datalog), routing would not converge deterministically or not converge at all. See § 4.1 for examples.

Because of these limitations, we rewrote Batfish's control-plane model as imperative code. Stage 1 of Batfish still parses configuration text into a vendor-intermediate format, but it now uses a Java data structure rather than Datalog facts. With the elimination of Datalog, provenance tracking for counterexamples was no longer automatic, and we thus also had to develop new mechanisms for that functionality.

Lesson 2: BDDs are great for data plane analysis

As we scaled Batfish to large networks, the performance of data plane verification also became a bottleneck. Our optimizations of NoD and Z3 SMT queries did not yield needed levels of performance. We eventually developed a new verification engine based on binary decision diagrams (BDDs), which represent boolean functions as decision DAGs. The BDD-based engine became the production implementation for data plane verification in Batfish in early 2018.

We discovered that BDDs can resolve the long-standing tension between custom and general backends for data plane verification [62]. Tools based on general backends [43, 45, 56, 57], such as Datalog or SMT, are easier to develop—heavy lifting is done by another engine. But they perform worse because many domain-specific optimizations can be difficult or impossible to express. On the other hand, "vertically integrated" tools based on custom encodings [33, 61, 62] perform well but are harder to develop and extend. Adding a new data plane function (e.g., NAT) requires invasive changes to both the data-plane encoding and the underlying engine. For example, adding packet transformations to the original Atomic Predicates [61] tool required development of an entirely new theory [62].

Our new engine, which uses a BDD-based dataflow analysis (§ 4.2), resolves this tension because (1) it can easily encode domain-specific optimizations, including topology and path related ones that are hard for SMT solvers [5]; (2) the number of BDD variables is largely independent of the network size; and (3) even advanced data plane functions, such as NATs and firewalls, tend to have compact BDD representations.

Lesson 3: Analysis fidelity is hard but not why you think

Batfish uses a model of how devices interpret configuration. This approach scales better than running actual device OS and routing

software [38]. However, it runs the risk of model inaccuracies and incorrect analysis. This risk is commonly attributed to different minor versions of the router software potentially processing configuration commands differently because of bugs or by design. Batfish may not know the version based on configuration text alone. We rarely encountered analysis fidelity issues due to this reason.

Instead, analysis fidelity issues arose mostly from *undocumented* semantics of router software. Though there are many commonly used configuration idioms, and even vendor-recommended templates, there are also many ways to deviate from these idioms, intentionally or accidentally. A simple example: What should happen to incoming routing announcements when a BGP neighbor is configured to use a route map that is not defined anywhere? This situation and many others are not documented by router software vendors, but there is a long tail of such situations that must be handled properly to correctly support real-world configurations.

Initially, we addressed such issues in an ad hoc manner as they arose, but by 2019 we were proactively testing deviations from standard configurations in an emulator (§ 4.3). Our approach does not *guarantee* that models match reality, but it has significantly reduced analysis fidelity issues. We also found that 100% fidelity is not required for Batfish to be useful. Even buggy models can find serious errors because there are classes of analyses (see Lesson 5) that are less likely to be impacted by modeling bugs.

Lesson 4: Usability is hard for reasons you think and then some

Based on experiences with software verification [9, 51], we expected usability challenges in making network verification accessible to engineers on the ground. Indeed, we faced some similar challenges, such as the absence of formal specifications. But we also faced three classes of challenges unique to networking.

1. Unclear invariant semantics. Network verification offers a different abstraction than tools like `traceroute` and `ping`, which are more familiar to network engineers. These other tools deal with individual concrete endpoints (sources, destinations), packets, and paths. Batfish originally accepted properties as first-order logic formulas that quantify over these entities. Requiring network engineers to write and understand such formulas was not practical. Therefore, we provided queries that could be richly parameterized with endpoints, packets headers, and paths based on intended invariants. But even simple invariants expressed in English can be highly ambiguous. "Set A of hosts should reach set B of hosts" could mean any of: all A hosts should reach all B hosts; some A host should reach all B hosts; all A hosts should reach some B host; some A host should reach some B host. Possibilities explode combinatorially if we also factor in packet headers and paths. Such ambiguities cause incorrect usage and confusion.

2. Uninteresting violations. Verification tools analyze all possible packet headers and paths. This completeness enables strong guarantees but also causes usability issues if flagged violations are uninteresting to users. In real networks, we encountered situations where a specified property was violated even though the network was configured correctly. For a simple query like "host A should reach service B," this could happen in a few ways: (a) when A sends packets with spoofed source IPs, which are then dropped at the access switch; (b) when A sends packets with low-numbered (privileged) source ports, which are then dropped at the firewall that protects B; (c) when A sends TCP packets with flags that indicate a response to a SYN from B, which the firewall blocks because B is not expected to send such packets. None of these may be an interesting violation for the user.

One could view the root issue as poor specification of the property. However, real networks have many such behaviors that are either not in the configuration (and caused by default device behaviors) or are in the part of the configuration that the user does not manage. It is unreasonable to expect network engineers to know and specify exact properties that preclude all uninteresting cases.

3. Explaining violations. Batfish originally provided one counterexample packet for violated properties that was picked by the SMT solver (randomly) from the violating header space. This approach has drawbacks. The counterexample does not indicate why it was picked (e.g., is the packet's source IP at fault or its source port?), lacks context (e.g., the specific source IP is not as meaningful as saying the source IP was in a user-defined prefix list), and can be confusing (e.g., the solver might pick 1.1.1.1 as the source IP, which cannot occur in their network). We also tried communicating the set of all packets that violate an invariant, but we found that it was difficult to do so in a way that does not overwhelm users but does lead them to the root configuration error.

We have not found a silver bullet for usability challenges, but, through trial and error we have discovered helpful techniques (§ 4.4), including creating narrowly scoped queries, applying reasonable defaults to analyses, and carefully selecting violation examples.

Lesson 5: Deep configuration modeling has many applications

The original goal of Batfish was analysis of network forwarding, but we soon found that network engineers wanted the tool to check many other configuration properties. These include checking: configuration settings (e.g., NTP servers), compatibility of BGP configuration across neighbors, whether all referenced routing policies are defined, uniqueness of IP addresses in the network, whether an access control list (ACL) attached to an interface allows a certain packet, and so on. Custom tools exist for some of these applications, but Batfish's detailed model of network configurations, which is a prerequisite for data plane generation, made it easy to support all these applications in one tool.

Most of these analyses are *local*, typically involving only a small portion of configurations of one or two nodes, and offer unique benefits. Users can easily localize identified errors. If a missing route-map results in bad forwarding, it is much easier to find this error by checking for undefined route-maps than by debugging based on the counterexample to a data plane verification query. As a complement to end-to-end forwarding guarantees, these analyses also help network engineers directly ensure that specific parts of the configuration are correct. This ability is particularly useful for management plane configuration (e.g., NTP servers), which does not directly impact forwarding.

Engineers also started using Batfish in workflows other than proactive configuration analysis. These workflows, described in

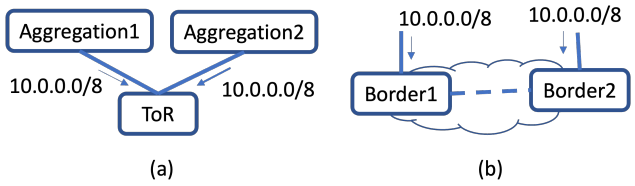

Figure 1: Two common network design patterns with non-deterministic convergence. (a) The aggregation switches export a route to the ToR (top-of-rack) switch, which uses the first one received (multipath is off). (b) Border routers prefer routes via each other to keep paths for certain prefixes internal, if possible. When both get such a prefix from outside, the network converges based on which receives the prefix first and exports it to the other.

§ 5, emerged because of limitations of the infrastructure in their organizations and of other testing tools.

4 DESIGN OF NEW COMPONENTS

We now describe the main elements of Batfish's new design.

4.1 Data plane generation

The new data plane generation engine in Batfish differs from the original in three main ways.

4.1.1 Imperative evaluation. The new engine is fully imperative and encodes the route exchange logic in custom Java code that runs a fixed-point computation. This aligns with supporting arbitrarily complex control plane models and enables Batfish to support a wide range of complex features (e.g., BGP add-path) and new protocols (e.g., EIGRP). It also allows us to control intricate dependencies between control plane and data plane state without having to encode them in Datalog. For example, the establishment of a BGP session between two peers depends on a successful TCP connection, which can be prevented by misconfigured ACLs or NAT (network address translation). We thus need to re-evaluate the viability of such sessions at key points in the data plane generation using partial data plane state.

Our imperative model lets us easily implement optimizations, such as allowing IGP protocols to converge prior to beginning BGP computation. We can also speed up the computation by introducing high levels of parallelism.

4.1.2 Optimized, deterministic convergence. Imperative evaluation by itself does not suffice because there are BGP networks for which it does not always converge or does not converge to a unique state. Figure 1 shows two routing patterns with non-deterministic convergence. With uncontrolled parallelism, where all nodes exchange routes with all their neighbors in the same iteration, simulating such networks triggers pathological cases where equally good routing advertisements trigger unnecessary re-computation. Or worse, routers get stuck in a re-advertisement loop because they proceed in lockstep, like the border routers in Figure 1b, where both: 1) export externally received 10.0.0.0/8 to each other; 2) both select the received internal paths (higher preference); 3) withdraw the direct external paths (exported in Step 1) because those are no longer selected; 4) select their own direct external paths because the internal paths were withdrawn; 5) export external paths to each other, which is the same as in Step 1, and the cycle repeats. This problem is not likely to occur in the actual network because both routers are unlikely to repeatedly and simultaneously export to each other.

To converge deterministically, we need to carefully limit parallelism without completely eliminating it (which hurts performance). Randomizing the processing order of routers can avoid the problem [15], but we also need consistent results across simulations to aid in debugging and presenting stable results to users. Paths should not change unnecessarily across network snapshots.

Batfish uses two convergence techniques. The first is a protocol-specific graph coloring. For each routing protocol, it computes the adjacencies, colors the graph [27], and allows only nodes of the same color to participate in the message exchange at the same time (for that routing protocol). This technique eliminates race conditions caused by neighbors exchanging routes given their partially converged state. Second, we add logical clocks [36] to our BGP RIB implementation, helping us to tie break routing advertisements based on arrival time, like routers do. This technique removes pathological re-advertisement loops. It does not, by design, force convergence on networks that do not converge in reality. Batfish detects and reports non-convergence of routing.

4.1.3 Optimized memory footprint. To scale to large networks, we need to optimize memory usage too. The classic fixed-point computation method—i.e., maintaining full RIB state for both the previous and current iteration to determine convergence—proved too expensive. We also tried the classic message-passing approach, where only current RIBs are stored but *incremental* updates (RIB deltas, i.e., routes added and removed) are processed by export policy and pushed onto queues to receivers. However, this approach keeps one distinct copy of each exported route per protocol neighbor, which is many more than the number of RIBs, until these queues are processed later by the receivers' import policy. When import policies can reject routes, it forces us to maintain considerable amounts of intermediate information in memory just to discard it later.

We settle on a hybrid approach with no queues. In addition to active RIBs, Batfish keeps only RIB deltas for the current and previous iteration. It can elide the queues because each receiver directly pulls from each protocol neighbor's RIB delta, combining the steps of processing the neighbor's export policy, processing the receiver's import policy, and merging accepted routes into its own RIBs. This lets Batfish efficiently determine when the computation converges by checking the emptiness of RIB deltas and reduce peak memory consumption to about the number of routes actually accepted by routers.

We also optimize the memory footprint of routing data itself. Batfish requires only a small fraction of the total memory capacity of the routers it simulates because it leverages the single Java process to intern common objects. The number of unique values for routing attributes is orders of magnitude lower than the total number of routes. Hence, we intern IP addresses, IP prefixes, BGP communities, and more complex routing attributes, such as BGP AS paths and BGP community sets (which also speed up equality

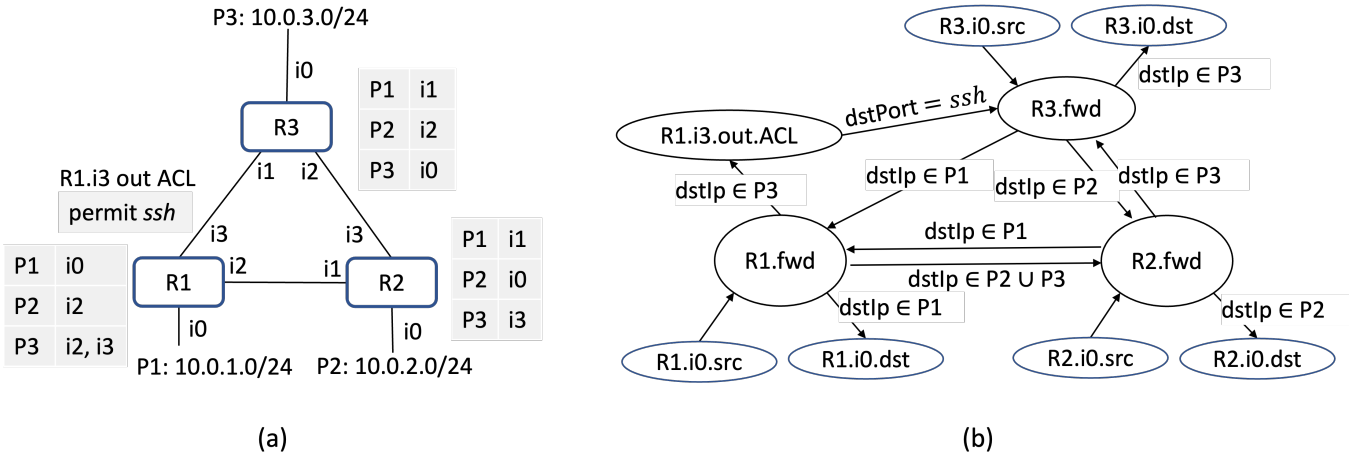

Figure 2: (a) An example network. Tables depict node forwarding tables (FIBs), mapping prefixes to outgoing interfaces. R1 uses multipath routing for P3 and has an outbound ACL on i3. (b) A simplified dataflow graph for the network. Edges are labeled with packet headers that can traverse them.

checks). We further reduce memory consumption with an awareness that *combinations* of these attributes are typically few. Many BGP advertisements that follow similar paths (e.g., because of multipath routing across data center network tiers) will share attributes such as administrative distance and BGP communities, AS path, and local preference. Moving 13 properties of a BGP route into a single interned object reduces the memory size of each route by 88 bytes, and there are typically 10x–20x fewer combinations of those properties than routes. This technique reduces memory consumption in typical networks by 50%.

4.2 BDD-based data plane analysis

The fundamental task of a data plane verification engine is to compute the set of packets that can reach particular nodes in the network. Like HSA [33], we use a form of dataflow analysis [35]: we build a dataflow graph that represents the paths through the network and then traverse the graph to compute the sets of packets that can reach each node. However, in Batfish we encode and manipulate sets of packets using BDDs, which provides a sweet spot between implementation complexity and performance. On one hand, unlike custom structures such as differences-of-cubes [33] and ddNF [10], we can leverage years of work on optimizing BDDs. On the other, unlike SMT solvers, we maintain complete control over the dataflow analysis and so can easily incorporate a host of extensions and optimizations. After reviewing the dataflow analysis with a small example, we describe our BDD-based representation; we then describe our main extensions and optimizations.

4.2.1 Overview of dataflow analysis. Consider the network in Figure 2(a), which shows the FIB (forwarding table) for each device. When R1 receives a packet destined to prefix P1, it forwards the packet out interface i0. It also has an outbound ACL on interface i3 that allows only ssh traffic.

Figure 2b shows the dataflow graph that Batfish builds for this network. It has three nodes representing the FIB lookups on each device and a node for the ACL on R1.i3. For each interface, there is also a source node for packets that it originates or receives from outside the network and a destination node for packets that it sinks or sends outside the network. The figure elides the source and destination nodes for internal interfaces (e.g., i2 for R1). It also elides a special "dropped" node to represent dropped traffic.

Edge labels in the dataflow graph indicate the set of packets that can flow along the edge, represented as a logical constraint, and are derived from FIBs and ACLs. For example, the edge from R1.fwd to D1 indicates that only traffic destined for an IP address in prefix P1 can traverse the edge, thereby accounting for the first row in R1's FIB. For real networks, the edge constraints are richer since they also encode the semantics of longest-prefix matching, ordering among ACL entries, and packet transformations.

The dataflow graph lets us compute the set of packets that can traverse between any pair of nodes. Following standard dataflow analysis, we start with the set of packets of interest at the source and iteratively traverse edges in the graph to update the set of packets that can reach each node, until we reach a fixed point. Suppose we want to compute all TCP packets that can enter the network via interface i0 at R1 (R1.i0.src in Figure 2b) and leave via i0 at R3 (R3.i0.dst). Initially, the set of packets at R1.i0.src is all TCP packets (whose IP protocol is 6), and the set of packets everywhere else is empty.

In every iteration, each node forwards its set along all outgoing edges, and receivers *union* the received set into their set to reflect everything reachable in prior iterations and everything newly reachable. In the first iteration, R1.fwd will receive the set of TCP packets from R1.i0.src and update its set accordingly. In the next iteration, R1.fwd will forward this set on its three outgoing edges, but the set is *intersected* with the set of packets denoted by the edge label in order to respect the constraints imposed by FIBs and ACLs. So R1.i3.out.ACL will receive the set of TCP packets that are destined for an IP in 10.0.3.0/24. This process inherently models multipath routing since the analysis traverses all paths. The process

continues until there are no more updates, at which point the set of packets at R3.i0.dst represents the answer to the original query.

4.2.2 Sets of packets as BDDs. Representing sets of packets explicitly is prohibitive, and various compact representations have been considered in the past [10, 33]. Batfish uses BDDs to represent the set of packets at each node as well as the edge labels in the dataflow graph. A BDD is a data structure that encodes a logical formula. We represent a set of packets as a logical formula over header bits and other information. For example, a BDD for the formula (dstIP ∈ P1) represents the set of packets destined for an IP address in prefix P1.

To understand how a BDD represents a formula over n bits, imagine a complete binary decision tree of height $n + 1$ (which has 2^n leaves). Each n-bit value corresponds to one path from the root to a leaf, taking the left (right) branch at depth i if the i-th bit is zero (one); each leaf node is labeled as true or false based on whether the formula is true for that value. A BDD optimizes this naive representation in two ways: (1) nodes at the same level that have identical subgraphs are merged, with all parents pointing to the single merged node (i.e., BDDs are DAGs, not trees); and (2) nodes where both branches point to the same child are deleted, with all incoming edges pointing to that child instead. Importantly, the resulting data structure is *canonical* for a given order of variables, and it can be substantially smaller than the original binary tree. BDDs also support efficient implementations of standard logical operations such as conjunction and disjunction, which respectively have the effect of intersecting and unioning sets of packets.

By expressing our computation using these operations, these facilities and various optimizations become available to us (e.g., we exploit canonicity to short-circuit full BDD traversals using identity-based operation caches). At the same time, the framework is flexible enough to facilitate extensions needed for real networks.

A key choice that we need to make is the BDD variable order, which dramatically affects the size of the resulting BDD. Batfish orders bits for the variables of an IPv4 packet using a simple heuristic. Subtree reuse is key to BDD efficiency, so fields that are filtered or transformed often should come first. We thus order header fields based on how frequently they are constrained, which leads to this order: Destination IP, Source IP, Destination Port, Source Port, ICMP Code, ICMP Type, IP Protocol, and finally less used fields, such as TCP Flags and Packet Length. Within a field, Batfish orders the bits with the most significant bit first.

Crucially, the number of BDD variables needed to represent a set of packets is primarily the number of bits in an IPv4 header, so it is independent of network design and size. As described below, we add more bits to model certain extensions, which can depend on network, but in practice they require very few additional variables. The real-world networks evaluated in § 6, which have 75–2735 devices, require only 0–6 additional BDD variables beyond the 261 (network-independent) variables that encode IPv4 header.

4.2.3 Extensions and optimizations. We now describe how we extend the basic dataflow analysis outlined above.

Packet transformations. Batfish supports precise reasoning about packet transformations such as NAT. We encode NAT rules as a BDD that has a second set of the variables corresponding to IP addresses and TCP/UDP ports (96 bits in total) and can thereby represent a function from the original packet to the transformed one. NAT edges intersect the BDDs for the input set of headers with the BDD for the NAT rule, then erase (existentially quantify) the input headers to get only the output headers, and finally remap variables in that BDD to those used to represent reachable sets (which are the same as the input variables in the NAT rule BDD). For efficiency, we implemented an optimized BDD operation to execute these three steps simultaneously.

Batfish encodes relationships between packets only on *edges* in the dataflow graph; at nodes, we always encode only sets of individual packets. This property keeps the BDDs in the reachability table small and enables the analysis to support arbitrarily many NATs without increasing the number of variables. In contrast, in SMT encodings [4, 45] each NAT doubles the variable count for the entire analysis, which significantly impacts performance. We interleave the variables for input-output packet pairs since a variable in the output packet tends to closely depend on the corresponding variable of the input packet.

Zone-based firewalls. Unlike ACLs, firewalls can filter packets based not just on headers but also on incoming and outgoing zones, where a *zone* is a set of interfaces. We model these using additional BDDs variables that represent incoming and outgoing zones. These variables are set to record the zone as the packet enters the firewall, are tested in subsequent edges that encode the firewall behavior, and are erased when the packet exits the firewall. Since the constraints are local to parts of the forwarding graph that encode a single firewall, we can reuse the same BDD variables to model zones on different firewalls. The number of variables needed to model firewalls for the entire network is logarithmic in the maximum number of firewall zones on any device, and in practice we have never needed more than four bits.

Stateful devices and bidirectional reachability. *Bidirectional reachability* reasons about packet sets that can make the round trip from A to B and back. In the presence of stateful behavior (e.g., NAT, firewalls), it is not the same as the intersection of two directions because what is permitted in the reverse direction depends on the state setup in the forward direction. (Most scalable data plane verification tools [33, 45, 62] do not analyze bidirectional reachability.) To analyze it, we first do a forward dataflow analysis, after which the reachable sets at nodes for stateful devices represent all firewall sessions that could be installed. We then instrument the dataflow graph by adding constraints to existing edges and inserting new ones to represent the session "fast path" for matching return traffic, and we then run the analysis in the other direction.

Waypoint queries. Batfish supports *waypoint-based queries* as well, where the goal is to ensure that paths between nodes traverse (or do not traverse) certain other nodes. We introduce variable(s) in packet sets that denote whether waypoint(s) of interest have been traversed. Initially unset, these bits are set when a waypoint is traversed during propagation. Though in principle waypointing constraints could be as complex as the size of the network, the typical verification has the form "all traffic from the Internet to a Webserver must traverse a Firewall" and requires only 1 bit.

Graph compression. Many nodes in the dataflow graph are simple, i.e., they have only one incoming or outgoing edge and do

not transform packets. Despite their simplicity, these nodes induce many edges in the graph, which slows down the analysis. We implemented an optimization that identifies and deletes these nodes. When a node is deleted, each pair of an in-edge from tail T and out-edge to head H is replaced by a single edge from T to H, representing the composition of those edges.

Query-based specialization. The dataflow analysis generally moves forward from sources to all destinations and dropped nodes, but a query may have a narrower interest. We implemented optimizations that, given a query, prune the graph. If a query is only for specific sources (or destinations), other sources (respectively, destinations) and corresponding edges are removed. Graph pruning is fast (no BDD operations), and it creates additional opportunities for graph compression. We also optimize how we walk the graph based on the query. For instance, if the query is interested only in traffic accepted at a single device, we walk the graph backwards from the destination toward the sources, propagating packet sets from edge head to tail. Backward propagation is achieved by "reversing" the original BDD. The reverse BDD represents what packet set could have arrived at the tail given a packet set at the head; it saves us from walking the edges that do not lie on the destination's forwarding tree.

4.3 Analysis fidelity

Two testing frameworks help us improve analysis fidelity.

4.3.1 Validation against ground truth. To make Batfish's analysis robust to a range of configuration patterns, we use a validation framework with real device software and emulation engines like GNS3 [22]. Our framework has the following workflow: 1) Create small networks (i.e., labs) exercising the features of interest using recommended configuration and possible deviations. 2) Collect device configurations and runtime state from the network, such as `show` commands for interfaces, routes, and VRFs as well as ping and traceroute data. 3) Validate that, given the collected configurations, the Batfish model aligns with the collected runtime state.

Such a validation framework has several advantages. It lets us leverage network engineer experts to create labs without their needing to know Batfish internals. When on-boarding new networks, we can re-use steps 2 and 3 to validate Batfish models against data from the live network. Data from labs and live networks goes into a repository, and step 3 is run daily on all networks, reducing the risk of regressions as Batfish code evolves.

The level of protection against modeling gaps this framework provides depends on the likelihood of encountering an untested deviation. Today, we use the expertise of network engineers to create possible deviations, which has worked well judging by the low occurrence of users reporting modeling issues. In the future, we will automate this process to increase coverage.

4.3.2 Differential engine testing. Batfish has two independent forwarding analysis engines: the BDD-based engine (described in § 4.2) and a traceroute engine that operates on concrete (not symbolic) packets. Validating that such engines produce identical results is instrumental in uncovering modeling bugs.

To validate equivalency, we perform two tests, one in each direction, with one engine serving as the "verifier" for the other. We first execute reachability queries for each final location in the network (with location roughly corresponding to node/interface pair), collecting a set of start location and headerspace tuples. Then, for each tuple, we pick a representative packet from the headerspace and run the traceroute engine to ensure that the final location and packet disposition match that reported by the reachability engine.

In the other direction, we first walk over each node's FIB, and for each entry, we randomly choose a packet with a destination that matches the entry's prefix. Then, we run the traceroute engine to find the terminal packet location and disposition. Finally, we run the reachability analysis from the terminal location and check if the set of computed start locations contains the original start location of the traceroute.

Routinely running this cross-validation enabled us to detect incorrect handling of complex features in production networks, such as policy-based routing, cross-VRF leaking, and session-based NATs. Similar to forwarding, Batfish has symbolic and concrete engines for analyzing ACLs and routing policies. We plan to develop differential testing engines for these analyses, as well.

4.4 Improving usability

We use three techniques to improve the usability of Batfish.

4.4.1 Specialized queries. As noted earlier, general-purpose queries that can be parametrized flexibly are hard to use because they lead to semantic ambiguities. Batfish now wraps the underlying general mechanisms with highly task-specific queries. Checking if a service endpoint is reachable from its intended client locations is a separate query from checking if a service cannot be reached. Query specialization also helps with scoping the search and picking examples, which we describe next.

4.4.2 Scoping the default search space. To minimize uninteresting violations, unless explicitly overriden by the user, Batfish automatically scopes the search space of headers and locations based on the query. Consider an "all pairs" reachability query to check that all hosts connected to the network can reach each other and the external world. For this query, we limit the search space of starting and end locations (interfaces) to those that face hosts or the external world because inter-router interfaces are commonly not of interest (and blocked). We identify host-facing interfaces using heuristics based on interface IP address and prefix-length, configured protocols, and whether we have the remote end of the link. We also limit the set of source and destination IPs to those that can likely originate or sink at those interfaces. Task-specific queries help devise reasonable defaults (e.g., defaults for a reachability-oriented query differ from those for a security-oriented one), and such defaults work well for a large majority of cases.

4.4.3 Positive examples and example selection. Example packets that violate an invariant are key to helping users understand the violation. However, this has limitations, as described earlier. To aid in understanding, instead of showing only the counterexample, Batfish also shows a positive example, i.e., a packet that does not violate the property. Contrasting the two examples is helpful. If they differ only in source ports, the source port of the counterexample is problematic.

Batfish picks examples (positive or negative) carefully to match what is likely for the network. The source and destination IPs of the packet should match where the packet starts and ends, even if specific addresses are irrelevant (e.g., because filtering is based on ports), and common protocols (e.g., TCP) and applications (e.g., HTTP) are prioritized. BDDs help to select positive and negative examples quickly by intersecting the answer space with preferences constraints (also encoded as BDDs). Finally, we annotate example packets with as much context as possible, such as the routing and ACL entries that they hit along their path.

5 PRIMARY USE-CASES IN THE FIELD

As Batfish matured, network engineers began to use it in a variety of ways beyond its originally designed use-case. We discuss the three primary use-cases that have emerged. "Use-case" here refers to how and when Batfish is used as part of network management; another dimension of usage is analysis types (§ 3); users employ multiple analysis types within each use-case.

5.1 Proactive validation

Proactive validation, where configuration changes are analyzed before being pushed to the network, was the original use-case for Batfish. For this, two distinct workflows have emerged depending on how the organization generates and vets configuration changes.

5.1.1 Automated workflow. Some organizations automatically generate configuration changes from high-level metadata (such as topology and IP addresses) and then review those changes. Sometimes called *network CI (continuous integration)*, this process is similar to software CI pipelines. Batfish is invoked as part of this process.

Unique to this workflow are the types of analysis used. Engineers are less likely to use linting or consistency checks (e.g., Are all referenced route maps defined? Are NTP servers consistent across all devices?). With auto-generated configurations, the chances of such errors are low. On the other hand, engineers are more likely to use checks that validate end-to-end forwarding behavior or behavior of ACLs. Creating such checks requires engineers to know the intended network behavior, a more likely scenario in organizations with automated workflows.

5.1.2 Manual workflow. Organizations that automatically generate and test configuration changes are a minority today [18]. Most invoke proactive validation using custom scripts or a browser-based UI. They are more likely to use simpler analyses (e.g., consistency). For behavior validation, they tend to use checks that target the change being made (e.g., a new BGP session should come up, or traffic should not traverse a link that is being shut for maintenance).

The most interesting anecdotes about the value of Batfish emerge from this setting. In one instance, an engineer was testing a configuration change to switch how the network connected to its transit provider. They initially thought that only two border devices needed changing. But after testing the change with Batfish by analyzing packet forwarding and routing tables, they realized that additional devices needed updates because of unexpected interactions in routing policies across devices. After iterating a few more times, they ultimately discovered that the configuration of ten devices needed updates for the change to be correct. The engineer told us that Batfish saved them days of frustration and configuration debugging (which is difficult to do on a live network).

5.2 Continuous validation

In this use-case, network engineers periodically analyze the latest snapshot of deployed network configurations. Doing so does not prevent errors from reaching the network, but it flags errors introduced since the last run, hopefully before the error impacts application traffic. This use-case is popular because getting started is easy: one needs only periodic snapshots of network configurations, which most organizations already have. It does not need infrastructure for gating configuration changes based on validation. We find that continuous validation is used even in organizations that have gating configuration infrastructure because it can detect errors introduced by out-of-band changes, which are common inside most networks.

An interesting aspect of real-world network operations that we learned via this use-case is that completely error-free configurations are generally not a high-priority goal. When Batfish is used in a network whose configurations have evolved over the years, it invariably finds errors and inconsistencies. However, unless these issues pose a serious risk, engineers do not fix them urgently; the act of fixing imposes overhead and risks (e.g., another error may be introduced, or the router may not reset properly). Fixing such issues becomes a low-priority, background project. Batfish helps monitor network progress and ensure that *new* errors are not introduced. Further, we see a greater willingness to go fix new errors immediately.

5.3 Design validation

An unexpected use-case of Batfish has emerged: it is used as a design tool. This is due to its ability to reason about configurations offline in the absence of a corresponding network. Network engineers who are designing new networks use Batfish as a first gate for validating newly developed configurations. Before Batfish, they would use emulations or a physical lab environment, which require hardware or software that may not be available (yet). These environments are also hard to scale to the size of the real network, and root causing errors is difficult. After Batfish certifies the configurations, the engineers may optionally test them in a small-scale emulation environment and then on the real network. Since the network is not yet carrying real traffic, Batfish's key added value is not outage prevention, but accelerating development and testing by days or even weeks.

A use-case that lies between validating new designs and validating daily configuration changes is validating large-scale refactoring of network design. Common refactoring operations include compressing large ACLs by removing redundant, no-longer-relevant, or unreachable entries (e.g., see [12]) or changing the routing design from OSPF to BGP. As with new designs, engineers use Batfish to test refactored configurations before using other forms of testing.

When Batfish is used as a design tool, we observe a tendency to modify configurations in order to make validation easier or faster. This ranges from minor configuration elements (e.g., preferring Batfish-supported syntax) to major aspects of the design. One network engineer discussed disfavoring iBGP full meshes because

Batfish takes longer to generate the data plane for such networks. Their reasoning, which is correct in this case but may not be in general, [1] was that if Batfish takes longer to compute the data plane, the convergence time of the real network will be longer, as well. As automated testing became popular for software, it became more likely for engineers to develop it in a way that aids testing [17]. We are starting to see a similar trend with configuration testing.

6 PERFORMANCE BENCHMARKING

We evaluate the current version of Batfish by comparing it to the original Batfish and benchmarking it on 11 real networks. We focus on the performance of parsing, data plane generation, and verification. We point to Batfish's burgeoning usage as indicating that our usability and fidelity analysis techniques have been effective.

Table 1 lists the 11 networks that we study. They are of diverse types and sizes and use different vendors and configuration features. To our knowledge, ours is the first study of a verification tool on such a broad array of real networks.

6.1 Improvement over the original

We compare the performance of the current and original Batfish. We use the original Batfish code and NET1, a network also used in the original paper [16]. The original code does not support the configuration features of our other real networks. To evaluate data plane verification, we use the multipath consistency query. It checks if there is any flow (starting location and packet headers) in the network that is accepted along some paths and denied along others. It simultaneously reasons about all forwarding rules in the network and thus provides an appropriate benchmark for data plane verification. This experiment uses an Intel Xeon E5-2683 computer.

Figure 3 shows that the performance of current Batfish is notably better than the original. *Data plane verification sped up by 12x* because we replaced NoD and Z3 with a BDD-based engine. Users commonly run multiple verification queries on a network snapshot, so the absolute advantage of faster verification accumulates. *Data plane generation sped up by 1500x* because we replaced Datalog with an optimized imperative engine. Finally, parsing sped up by 37% because of better coding discipline. Parsing experts will appreciate the changes: elimination of semantic predicates; minimization of expensive adaptive prediction by shaping parser rules to make most decisions LL(1); and liberal use of lexer modes, which have replaced almost all usages of cache-disabling lexer predicates.

The net impact of these performance improvements is that Batfish went from an offline analysis tool that takes hours to run to one that can be run interactively or inside a CI pipeline for every configuration change. This represents a significant shift in how and how often it can be used.

6.2 Current performance

Table 2 shows the performance of current Batfish on all networks. These experiments use an Apple M1 Max Macbook.[2] To benchmark data plane verification, we study three tasks: (1) *full dataflow*: build and optimize the dataflow graph along which all possible packets can propagate from all sources used for subsequent queries; (2) *destination reachability*: compute which sources can reach a destination using which packets; and (3) *multipath consistency*. This query runs substantially faster for NET1 than the results in Figure 3 because it reuses the full dataflow result.

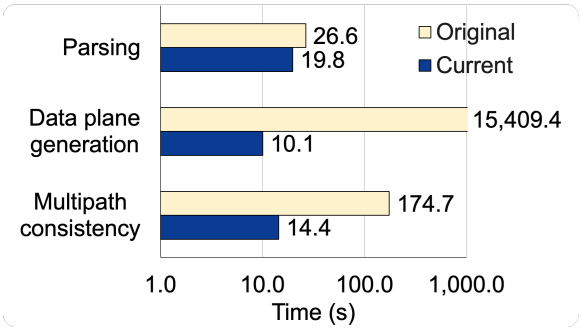

Figure 3: Performance of original and current Batfish for NET1. X-axis is log scale.

We see that all stages perform well. For NET11, which has 2735 nodes and nearly 16 million routes, the user can initialize the snapshot, generate the data plane, and run 100 destination reachability queries serially, and the analysis will finish in 5 minutes.

We also see that analysis performance depends only weakly on network size—network design and configuration features matter, as well. As a rule of thumb, DC networks perform better because they tend to have simpler policies. NET2 and NET3, both DCs, outperform the smaller NET1, a campus network. Similarly, NET8 (a DC) outperforms the smaller NET7 (backbone).

Finally, because configurations files are large, parsing time is a substantial component, as well, which is why we made parsing incremental in Batfish. Configuration files that have not changed since the last network snapshot are not parsed again. The figure shows the time to parse all files from scratch.

6.3 Comparison to the state of art

We place the performance of current Batfish in context with other state-of-art tools based on published benchmarks. Head-to-head comparison on identical data is difficult because the code for those tools is not available or the tools use different inputs (configurations versus FIBs).

For data plane generation, FastPlane [44] is the fastest tool to our knowledge. It works only for networks with monotonic routing, where routes once deemed non-best during route computation can never be best later. Monotonicity enables FastPlane to speed data plane generation by retaining only current best paths in memory and comparing new routes against this much smaller set. The authors show that FastPlane is speedier than the *original* Batfish by two orders of magnitude, while Batfish's current imperative engine, based on graph-coloring based parallelism and memory optimizations, is faster than the original version by three orders of magnitude. Batfish also works for non-monotonic networks, which all networks other than eBGP-only datacenters tend to be.

[1] We have not systematically validated conditions under which Batfish's convergence time can be used as a proxy for the real network's convergence time.

[2] The computer is different from the one in § 6.1 because we could not copy some networks' data to the other machine for confidentiality reasons, and the LogicBlox engine used by the original Batfish does not run on the M1 chip.

	Network type	Devices (links)	LoC (k)	Routes (k)	Device types	Routing protocols
NET1	Campus	75 (312)	156	69	Cisco IOS, Juniper	OSPF, eBGP, iBGP
NET2	DC	124 (1,690)	178	151	Cisco NX-OS	eBGP, iBGP
NET3	DC	139 (6,306)	193	262	Arista, Cisco ASA	eBGP, iBGP
NET4	Paired DCs	205 (1,278)	335	35	Cisco IOS/NX-OS, Juniper	IS-IS, OSPF, eBGP, iBGP
NET5	DC	211 (1,280)	338	301	Cisco IOS, F5, Juniper	IS-IS, OSPF, eBGP, iBGP
NET6	Hybrid cloud	217 (2,584)	41	14	AWS, Cisco IOS	eBGP
NET7	Backbone	244 (3,946)	254	774	Arista, Cisco IOS/IOS-XR/NX-OS	EIGRP, eBGP, iBGP
NET8	DC	268 (1,486)	469	165	Cisco IOS, F5, Juniper	OSPF, eBGP
NET9	Backbone+DCs	1,095 (17,100)	1,496	473	Arista, Aruba, Cisco IOS, Fortinet	OSPF, eBGP, iBGP
NET10	DC	1,283 (18,218)	7,016	180	A10, Check Point, Cisco IOS/IOS-XR/NX-OS	EVPN, OSPF, eBGP, iBGP
NET11	DC	2,735 (40,126)	8,651	15,947	Arista, Cisco NX-OS, Juniper	eBGP, OSPF

Table 1: Real networks that we study. In the network type column, "DC" refers to data center and "paired DCs" refers to two nearby data centers that provide backup connectivity to each other. "LoC" is configuration lines across all files, and "routes" is the total number of routes in the main RIB (which contains the best route to a prefix across all protocol instances) across all devices. Among device types, F5 and A10 are load balancers; Cisco ASA, Fortinet, and Check Point are stateful firewalls. In addition to the listed routing protocols, all networks have ACLs and static routes.

	Parsing (s)	DP gen (s)	Full dataflow (ms)	Dest reach (ms)	Multipath consistency (ms)
NET1	11.1	6.1	4,126.7	1.3	2,439.0
NET2	4.9	4.8	775.7	0.9	320.1
NET3	8.6	4.6	916.5	0.8	380.0
NET4	8.8	2.8	461.2	2.4	403.5
NET5	9.1	6.2	1,197.8	4.3	416.2
NET6	2.3	2.0	384.2	0.1	191.0
NET7	13.5	11.9	1,382.5	16.5	4,635.0
NET8	12.7	4.2	1,668.6	2.2	340.3
NET9	16.6	11.9	1,719.8	11.7	6,880.0
NET10	46.6	47.7	7,456.0	7.7	19,840.0
NET11	71.6	200.6	19,013.0	52.7	16,889.0

Table 2: Performance of current Batfish. "DP gen" is data plane generation, and "Dest reach" is destination reachability.

Data plane verification can be incremental or non-incremental. In the incremental case, forwarding rule updates atop a current data plane state snapshot are processed [23, 24, 32, 34]. This setting is relevant for networks where SDN controllers generate rule updates. In the non-incremental case, a full snapshot is analyzed [33, 45, 61, 62]. Batfish targets this setting, where the best performing tool to our knowledge is APT [62].

The largest network the APT authors study has 92 nodes. For this network, on an Intel Xeon E5-1650 processor, APT takes 335 seconds (not ms) to generate atomic predicates, which, like our full dataflow graph, is a one-time analysis that is reused by subsequent queries; further, APT takes 78 ms for destination reachability queries. In contrast, for NET2, which is 35% larger than APT's 92-node network, Batfish builds the dataflow graph and answers destination reachability queries almost two orders of magnitude faster. This large difference is unlikely the result of differences in experimental hardware.

7 OUTLOOK

The following challenges, if addressed, will make it easier to build, deploy, and use network verification tools like Batfish.

7.1 Automation woes

The dominant hurdle toward broader use of network verification is the lack of automation in most networks. Hyperscalers have sophisticated network automation, but the practice of networking in other organizations is starkly different. When network configuration changes are created, reviewed, and deployed manually, it is difficult to inject automated validation. In manual workflows, the exact configuration change may not even be instantiated until an engineer logs into the router and types it in. Before that, the artifacts that are created and reviewed are semi-structured at best. Using automatic validation here is a high-friction activity, undertaken only for high-risk changes.

Even in networks where automation exists, the diversity of frameworks makes it challenging to develop and deploy verification tools. Diversity in automation frameworks exists because how one gets configurations from devices, pushes changes, and monitors them varies across router vendors and platforms. Consequently, many networks have custom solutions atop a mix of methods. Standardizing the APIs of automation frameworks, akin to POSIX, will go a long way toward accelerating the adoption of validation tools.

7.2 Heterogeneous data plane pipelines

Heterogeneity of networking devices poses a challenge to tool development and maintenance, as well. Batfish tackles configuration syntax heterogeneity by translating a vendor-specific configuration into a general representation. It takes a similar approach to data plane pipelines. The original Batfish design had a simple, 3-step data plane pipeline: packets go through the ingress interface ACL, then through forwarding table lookup, and then the egress interface ACL. To support more devices, we needed more steps, such as stateful firewalls, source NAT (network address translation), and destination NAT. Unfortunately, vendors do not use the same order for these steps. Some may NAT before routing lookup and some after, some may firewall based on pre-NAT headers and some on post-NAT headers, etc. Whenever we encountered a different behavior, we generalized the vendor-independent pipeline to include more steps so that any device's behavior could map to a subset of steps in that general pipeline. Data plane analyses operate off of this general pipeline.

Over time, this general pipeline has become complex, and every time it changes, we must update all analyses that use this pipeline as input. This complicates the addition of new features and creates a risk of analysis bugs. A language-based approach, instead of our current data-model based one, offers a potential solution [8]. An intermediate language could express the data plane pipeline functionality, and each device's pipeline would be a program in the language. Analysis tools could then more easily analyze a pipeline of programs in that language.

7.3 Usability and performance

Our techniques to improve the usability and performance of Batfish made a significant difference, but there is room for further improvement. For usability, a key challenge is that Batfish normalizes configurations to support many vendor-specific languages. This lets it easily analyze many vendors, but it also loses important source-level information that can help users debug errors. Compilers face a similar problem since they translate code into different lower-level intermediate representations and optimize these representations. A common compiler mitigation technique includes metadata with each intermediate-level instruction that contains information, such as the corresponding source-level locations, and updates this metadata through transformations and optimizations [39]. These kinds of techniques would likely prove helpful for tools like Batfish.

To improve performance, a promising opportunity is to make data plane generation even faster via incremental computation. Instead of cold starting the simulation, one could start with the last computed state. This would be helpful because changes in topology or configuration across successive network snapshots are small. For correctness, we must carefully determine what needs recomputation when some aspect of the network changes.

8 RELATED WORK

This paper is inspired by researchers who have shared experiences of transitioning tools based on formal methods to practice [9, 11, 25, 51]. Their experiences helped us anticipate challenges, such as the lack of formal specifications, heterogeneity of language constructs across organizations, and how users view bug-finding tools. We also faced challenges unique to networking, and we hope their description will improve future networking analysis tools.

Hoyan [63] and RCDC [26] developers shared their experiences of deploying network verification tools in cloud networks. Like Batfish, Hoyan focuses on configuration analysis, and some of the challenges it faced, such as analysis fidelity, were similar, too. However, our usage context—analyzing arbitrary third-party networks instead of a first-party network—markedly differs from these prior works, which changes the nature of our challenges and solutions. We cannot rely on intimate knowledge of network design to improve scalability [26] or on continuous, live data from the network to improve analysis fidelity [63]. At the same time, we must support a broader range of vendors as well as users who are not deeply familiar with the tool's internals.

Our new data plane generation engine builds on past work in this domain. Several tools, including C-BGP [52] and FastPlane [44], simulate network routing. Our engine scales two orders of magnitude more than C-BGP, which experimented with networks with tens of nodes. Unlike FastPlane, which works only for datacenter networks with monotonic routing, our engine works for any network. Most real networks that we have encountered are non-monotonic.

There have been many data plane verification tools [23, 24, 26, 32–34, 45, 49, 56, 57, 61, 62, 65]. Our graph walking is similar to HSA [33], but we use BDDs instead of a custom data structure (called differences of cubes). BDDs let us use a general backend and easily implement a range of optimizations and rich data plane functions. Bonsai [7] and Atomic Predicates [61] inspired our use of BDDs, though they use BDDs for different purposes (route map modeling and equivalence class inference). BDDs have recently been used to implement rich network forwarding, as well [28].

9 CONCLUSIONS

Batfish has evolved from a research prototype to a mature tool for analyzing network configuration. While its original goal and architecture stood the test of time, many of its underlying techniques had to be revamped to address the scalability and usability challenges presented by complex, real-world networks. This evolution offers many lessons, including the limitations of Datalog and the effectiveness of binary decision diagrams (BDDs) for network analysis. Replacing Datalog with an imperative routing simulation engine and a BDD-based data plane verification engine improved performance by three orders of magnitude; a host of additional techniques to improve analysis fidelity and usability enabled Batfish's use inside a diverse range of real networks.

Acknowledgments

Batfish could not have been developed without contributions from many people beyond this paper's authors. We are thankful to all Intentionet employees, especially those who contributed to the analytical aspects of Batfish, including Spencer Fraint, Corina Miner, Samir Parikh, Harsh Verma, Chirag Vyas, and Yifei Yuan. We are also grateful to Batfish users and its developers among the broader open source community.

This work was supported in part by NSF grants CNS-1901510, CNS-2007073, and FMiTF-2219863.

This work does not raise ethical issues.

REFERENCES

[1] Anubhavnidhi Abhashkumar, Aaron Gember-Jacobson, and Aditya Akella. 2020. Tiramisu: Fast Multilayer Network Verification. In *NSDI*.

[2] Akers. 1978. Binary Decision Diagrams. *IEEE Trans. Comput.* (1978).

[3] E.S. Al-Shaer and H.H. Hamed. 2004. Discovery of Policy Anomalies in Distributed Firewalls. In *INFOCOM*.

[4] John Backes, S. Bayless, Byron Cook, C. Dodge, Andrew Gacek, A.J. Hu, T. Kahsai, B. Kocik, E. Kotelnikov, J. Kukovec, Sean McLaughlin, J. Reed, Neha Rungta, J. Sizemore, M. Stalzer, P. Srinivasan, P. Suboti, Carsten Varming, and B. Whaley. 2019. Reachability Analysis for AWS-based networks. In *CAV*.

[5] Sam Bayless, Noah Bayless, Holger H. Hoos, and Alan J. Hu. 2015. SAT Modulo Monotonic Theories. In *AAAI*.

[6] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. 2017. A General Approach to Network Configuration Verification. In *SIGCOMM*.

[7] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. 2018. Control Plane Compression. In *SIGCOMM*.

[8] Ryan Beckett and Ratul Mahajan. 2020. A General Framework for Compositional Network Modeling. In *HotNets*.

[9] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. 2010. A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World. *Commun. ACM* (2010).

[10] Nikolaj Bjørner, Garvit Juniwal, Ratul Mahajan, Sanjit A. Seshia, and George Varghese. 2016. ddNF: An Efficient Data Structure for Header Spaces. In *Haifa Verification Conference (HVC)*.

[11] Cristiano Calcagno, Dino Distefano, Jeremy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. 2015. Moving Fast with Software Verification. In *NASA Formal Methods*.

[12] Antonio Ceseracciu. 2020. Safe ACL Change through Model-based Analysis. https://tech.ebayinc.com/engineering/safe-acl-change-through-model-based-analysis/.

[13] Seyed K. Fayaz, Tushar Sharma, Ari Fogel, Ratul Mahajan, Todd Millstein, Vyas Sekar, and George Varghese. 2016. Efficient Network Reachability Analysis Using a Succinct Control Plane Representation. In *OSDI*.

[14] Nick Feamster and Hari Balakrishnan. 2005. Detecting BGP Configuration Faults with Static Analysis. In *NSDI*.

[15] Sally Floyd and Van Jacobson. 1993. The Synchronization of Periodic Routing Messages. *SIGCOMM Computer Communication Review (CCR)* (1993).

[16] 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 *NSDI*.

[17] R.S. Freedman. 1991. Testability of software components. *IEEE Transactions on Software Engineering* (1991).

[18] Damien Garros. 2020. The State of Network Operation Through Automation / NetDevOps Survey 2019. https://blog.networktocode.com/post/state-network-operations-netdevops-survey-2019/.

[19] Aaron Gember-Jacobson, Raajay Viswanathan, Aditya Akella, and Ratul Mahajan. 2016. Fast Control Plane Analysis Using an Abstract Representation. In *SIGCOMM*.

[20] Aaron Gember-Jacobson, Raajay Viswanathan, Aditya Akella, and Ratul Mahajan. 2016. Fast Control Plane Analysis Using an Abstract Representation. In *SIGCOMM*.

[21] Nick Giannarakis, Devon Loehr, Ryan Beckett, and David Walker. 2020. NV: An Intermediate Language for Verification of Network Control Planes. In *PLDI*.

[22] gns3 [n. d.]. GNS3 | The Software that Empowers Network Professionals. https://www.gns3.com/. Retrieved February 2023.

[23] Dong Guo, Shenshen Chen, Kai Gao, Qiao Xiang, Ying Zhang, and Y. Richard Yang. 2022. Flash: Fast, Consistent Data Plane Verification for Large-Scale Network Settings. In *SIGCOMM*.

[24] Alex Horn, Ali Kheradmand, and Mukul R. Prasad. 2017. Delta-Net: Real-Time Network Verification Using Atoms. In *NSDI*.

[25] Marieke Huisman and Raúl E. Monti. 2020. On the Industrial Application of Critical Software Verification with VerCors. In *Leveraging Applications of Formal Methods, Verification and Validation: Applications*.

[26] 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 *SIGCOMM*.

[27] Tommy R. Jensen and Bjarne Toft. 1994. Graph Coloring Problems.

[28] Theo Jepsen, Ali Fattaholmanan, Masoud Moshref, Nate Foster, Antonio Carzaniga, and Robert Soulé. 2020. Forwarding and Routing with Packet Subscriptions. In *CoNEXT*.

[29] Siva Kesava Reddy Kakarla, Alan Tang, Ryan Beckett, Karthick Jayaraman, Todd Millstein, Yuval Tamir, and George Varghese. 2020. Finding Network Misconfigurations by Automatic Template Inference. In *NSDI*.

[30] Jeff Kala. 2022. Developing Batfish. http://blog.networktocode.com/post/batfish-development-part1/.

[31] Michael Kashin. 2018. Network CI/CD Part 3 – Building a network CI pipeline with Gitlab, Ansible, cEOS, Robot Framework and Batfish. https://aristanetworks.force.com/AristaCommunity/s/article/network-ci-part-3.

[32] Peyman Kazemian, Michael Chang, Hongyi Zeng, George Varghese, Nick McKeown, and Scott Whyte. 2013. Real Time Network Policy Checking Using Header Space Analysis. In *NSDI*.

[33] Peyman Kazemian, George Varghese, and Nick McKeown. 2012. Header Space Analysis: Static Checking for Networks. In *NSDI*.

[34] Ahmed Khurshid, Xuan Zou, Wenxuan Zhou, Matthew Caesar, and P. Brighten Godfrey. 2013. VeriFlow: Verifying Network-wide Invariants in Real Time. In *NSDI*.

[35] Gary A. Kildall. 1973. A Unified Approach to Global Program Optimization. In *POPL*.

[36] Leslie Lamport. 1978. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM* (1978).

[37] Guyue Liu, Ao Li, Christopher Canel, and Vyas Sekar. 2021. Watching the Watchmen: Least Privilege for Managed Network Services. In *HotNets*.

[38] 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 *SOSP*.

[39] llvm-debug [n. d.]. Source Level Debugging with LLVM. https://llvm.org/docs/SourceLevelDebugging.html. Retrieved February 2023.

[40] LogicBlox, Inc. 2015. LogicBlox 4 Reference Manual. https://developer.logicblox.com/content/docs4/core-reference/html/index.html.

[41] Boon Thau Loo, Tyson Condie, Minos Garofalakis, David E. Gay, Joseph M. Hellerstein, Petros Maniatis, Raghu Ramakrishnan, Timothy Roscoe, and Ion Stoica. 2006. Declarative Networking: Language, Execution and Optimization. In *SIGMOD*.

[42] Boon Thau Loo, Joseph M. Hellerstein, Ion Stoica, and Raghu Ramakrishnan. 2005. Declarative Routing: Extensible Routing with Declarative Queries. In *SIGCOMM*.

[43] Nuno P. Lopes, Nikolaj Bjorner, Patrice Godefroid, Karthick Jayaraman, and George Varghese. 2015. Checking Beliefs in Dynamic Networks. In *NSDI*.

[44] Nuno P. Lopes and Andrey Rybalchenko. 2019. Fast BGP Simulation of Large Datacenters. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*.

[45] Haohui Mai, Ahmed Khurshid, Rachit Agarwal, Matthew Caesar, P. Brighten Godfrey, and Samuel Talmadge King. [n. d.]. Debugging the Data Plane with Anteater. In *SIGCOMM*.

[46] Joel McGuire. 2021. Developing Batfish. https://joelmcguire1.medium.com/building-a-new-ipam-system-using-netbox-and-batfish-ad1e91ff3503.

[47] Timothy Nelson, Christopher Barratt, Daniel J. Dougherty, Kathi Fisler, and Shriram Krishnamurthi. 2010. The Margrave Tool for Firewall Analysis. In *Large Installation System Administration Conference (LISA)*.

[48] oracle-path-analyzer [n. d.]. Oracle Cloud Network Path Analyzer. https://docs.oracle.com/en-us/iaas/Content/Network/Concepts/path_analyzer.htm. Retrieved February 2023.

[49] Aurojit Panda, Ori Lahav, Katerina Argyraki, Mooly Sagiv, and Scott Shenker. 2017. Verifying Reachability in Networks with Mutable Datapaths. In *NSDI*.

[50] Terence J. Parr and Russell W. Quong. 1995. ANTLR: A Predicated-LL (k) Parser Generator. *Software: Practice and Experience* (1995).

[51] Pieter Philippaerts, Jan Tobias Mühlberg, Willem Penninckx, Jan Smans, Bart Jacobs, and Frank Piessens. 2014. Software verification with VeriFast: Industrial case studies. *Science of Computer Programming* (2014).

[52] Bruno Quotin and Steve Uhlig. 2005. Modeling the Routing of an Autonomous System with C-BGP. *IEEE Network: The Magazine of Global Internetworking* (2005).

[53] Jorge Romero. 2020. Using Gitlab Runners in Network Pipelines. https://www.linkedin.com/pulse/using-gitlab-runners-network-pipelines-jorge-romero/.

[54] slack-batfish-general [n. d.]. #general. https://batfish-org.slack.com/archives/C8XKQNHAQ.

[55] slack-ntc-batfish [n. d.]. #batfish. https://networktocode.slack.com/archives/CCE02JK7T.

[56] Serena Spinoso, Matteo Virgilio, Wolfgang John, Antonio Manzalini, Guido Marchetto, and Riccardo Sisto. 2015. Formal Verification of Virtual Network Function Graphs in an SP-DevOps Context. In *Service Oriented and Cloud Computing*.

[57] Radu Stoenescu, Matei Popovici, Lorina Negreanu, and Costin Raiciu. 2016. SymNet: Scalable Symbolic Execution for Modern Networks. In *SIGCOMM*.

[58] Alan Tang, Siva Kesava Reddy Kakarla, Ryan Beckett, Ennan Zhai, Matt Brown, Todd Millstein, Yuval Tamir, and George Varghese. 2021. Campion: Debugging Router Configuration Differences. In *SIGCOMM*.

[59] Nathan Winemiller. 2021. Batfish Configuration Validation Testing. https://www.dropbox.com/scl/fi/3jw64qfnlfwqc0051brls/Batfish-Configuration-Validation-Testing.paper?dl=0&rlkey=bw8rjngcfsatfv5s6g2vmqytn.

[60] Xieyang Xu, Weixin Deng, Ryan Beckett, Ratul Mahajan, and David Walker. 2023. Test Coverage for Network Configurations. In *NSDI*.

[61] Hongkun Yang and Simon S. Lam. 2013. Real-time verification of network properties using Atomic Predicates. In *International Conference on Network Protocols (ICNP)*.

[62] Hongkun Yang and Simon S. Lam. 2017. Scalable Verification of Networks With Packet Transformers Using Atomic Predicates. *IEEE/ACM Transactions on Networking* (2017).

[63] 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 *SIGCOMM*.

[64] Lihua Yuan, Hao Chen, Jianning Mai, Chen-Nee Chuah, Zhendong Su, and P. Mohapatra. 2006. FIREMAN: A Toolkit for Firewall Modeling and Analysis. In *IEEE Symposium on Security and Privacy*.

[65] 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 *NSDI*.