Differential Network Analysis

Peng Zhang*, Aaron Gember-Jacobson♯, Yueshang Zuo*, Yuhao Huang*, Xu Liu*, and Hao Li*
*Xi'an Jiaotong University, ♯Colgate University

Abstract

Networks are constantly changing. To avoid outages, operators need to know whether prospective changes in a network's control plane will cause undesired changes in end-to-end forwarding behavior. For example, which pairs of end hosts are reachable before a configuration change but unreachable after the change? Control plane verifiers are ill-suited for answering such questions because they operate on a single snapshot to check its "compliance" with "explicitly specified" properties, instead of quantifying the "differences" in "affected" end-to-end forwarding behaviors. We argue for a new control plane analysis paradigm that makes differences first class citizens. Differential Network Analysis (DNA) takes control plane changes, incrementally computes control and data plane state, and outputs consequent differences in end-to-end behavior. We break the computation into three stages—control plane simulation, data plane modeling, and property checking—and leverage differential dataflow programming frameworks, incremental data plane verification, and customized graph algorithms, respectively, to make each stage incremental. Evaluations using both real and synthetic control plane changes demonstrate that DNA can compute the resulting differences in reachability in a few seconds—up to 3 orders of magnitude faster than state-of-the-art control plane verifiers.

1 Introduction

Networks are frequently in flux. Configurations are modified monthly, or even weekly: e.g., two large universities change up to 55 stanzas per router per month [27], and Facebook conducts an average of 12.5 changes per device per week in their backbone [41]. External peers update routes daily: e.g., four tier-1 ISPs each experience a median of ∼100K route updates per day [12]. Links and routers fail intermittently: e.g., a large online service provider's data centers have a median of 18.5 link outages per day [17], and the CENIC research network has a median of 38.5 outages per link per year [42].

Each change poses a risk of introducing catastrophic network outages [28, 31, 46]. To avoid outages, operators need to know whether prospective changes in the control plane (i.e., changes in configurations, external routes, or available links/routers) will cause (un)desired changes in end-to-end forwarding behavior. For example, would a link failure break isolation? Would a configuration change reduce reachability? Would an external route withdrawal degrade load balancing?

At first glance, existing verifiers [1, 4, 6, 7, 13–15, 18, 22–24, 26, 33, 37, 40, 43, 45, 48] seem to address this need. However, data plane verifiers [18, 19, 23, 24, 26, 33, 44, 48] cannot directly answer these questions, because they operate on changes in the control plane's output, rather than the control plane itself. Existing control plane verifiers [1, 4, 6, 7, 13–15, 22, 37, 40, 43, 45] are also ill-suited for this task, because of the following limitations: (1) they focus on checking a single control plane snapshot instead of differences between snapshots and (2) they require a list of properties to check, but it is difficult for operators to determine which properties may be affected by a change and hence need to be checked.

For the first limitation, a possible workaround is to apply a control plane verifier to both the old and new snapshots and compare the verifier's output. But analyzing both snapshots from scratch is wasteful, because many control plane changes have a limited impact on the data plane. For example, fewer than 100 forwarding (ACL) rules are changed for >80% (>90%) of configuration changes in the backbone network at a large university (Figure 1), and Steffen et al.'s experiments with 90 ISP topologies show that only one-third of link failures impact the forwarding path between a randomly chosen ingress node and destination prefix [40].

For the second limitation, a possible workaround is to check reachability (or other properties) for all pairs of end hosts. But checking all possible properties is wasteful, because the space of all properties is large [9] and the set of properties affected by a change is often small. Policy-mining tools [8, 9, 25] can help narrow the space of properties, but the set of inferred properties may still be large—e.g., Config2Spec returns over 3K properties for a national research and education (R&E) network with only 10 routers [9].

Since control plane verifiers can be quite inefficient for

assessing whether control plane changes cause changes in end-to-end behavior, we argue for a new control plane analysis paradigm which makes differences first class citizens. *Differential Network Analysis (DNA)* takes as input differences in configurations, external routes, and available links/routers, incrementally computes control and data plane state, and outputs the consequent differences in end-to-end behavior (e.g., reachability, waypointing, load balancing, etc.). This aligns with the small size/impact of many control plane changes and avoids duplicate and unnecessary computations.

To incrementally compute differences in forwarding behavior based on differences in the control plane, DNA breaks the computation into three modular stages—control plane simulation, data plane modeling, and property checking—and makes each stage "differential". In other words, each stage consumes differences (control plane changes, data plane changes, and forwarding graph changes, respectively), incrementally updates its network model, and produces differences (data plane changes, forwarding graph changes, and property changes, respectively). To achieve incremental computation for each of these stage, DNA leverages differential dataflow programming frameworks [2, 35], incremental data plane verification [48], and customized graph algorithms, respectively.

We implement a version of DNA that supports differential analysis of two widely-used routing protocols (BGP and OSPF) and widely important properties (reachability, waypointing, and load balancing). Our implementation of DNA is publicly released under an open source license. We evaluate DNA using both synthetic and real control plane changes, and demonstrate that DNA can compute differences in reachability induced by control plane changes in a few second—up to 3 orders of magnitude faster than state-of-the-art control plane verifiers [1,4,6]. Second-level verification time can enable on-the-fly checking of operator-proposed configuration changes, similar to syntax checkers integrated into most programming IDEs, as well as quickly validating automatically-generated changes due to dynamic control [30].

In summary, we make the following three contributions:

- We propose differential network analysis (DNA), a new paradigm that helps operators better understand the impact of changes in the control plane.
- We design and implement DNA based on recent advances in differential dataflow programming and incremental data plane verification, and apply optimizations to overcome their inefficiencies.
- We use both synthetic and real control plane changes to show DNA computes consequent differences in end-to-end behavior up to 3 orders of magnitude faster than state-of-the-art control plane verifiers [1,4,6].

2 Motivation

In this section, we discuss in detail why invoking a control plane verifier before and after a change, and for all inferred properties, is an inefficient way to assess whether changes in the control plane cause changes in end-to-end behavior. In particular, we highlight: (1) the prevalence of small control plane changes, and (2) the difficulty of identifying which properties may be impacted by a change.

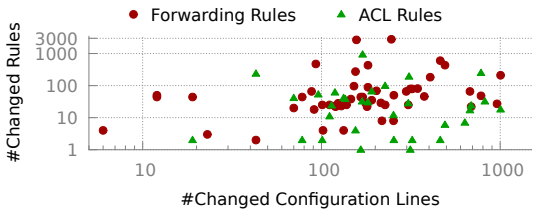


Figure 1: The number of changed (forwarding and ACL) rules, and the number of changed configuration lines for the backbone network at a large university.

2.1 Control plane changes are often small

Control plane verifiers operate on full snapshots of the control plane, but the delta between snapshots is often small, which leads to significant amounts of unnecessary re-computation.

Configuration changes. Several prior studies have examined router configurations across various organizations/networks and found that changes are often small: e.g., two large universities each change (on average) $\leq$20 lines of configuration at the same time [36]; changes in Facebook's backbone and data center networks impact an average of 157 and 738 lines of configuration, respectively, which is relatively small compared to the scale of these networks [41]; and in 75% of the networks operated by a large online service provider, the median change includes only three devices [16]. Consequently, we expect control plane verifiers' inputs and outputs to be similar before and after such changes.

To validate this hypothesis, we analyze 3 months of configuration changes from the backbone network at a large university [36]. The network has 28 routers and 50 links and runs OSPF. On average, the network has $\sim$75K total lines of configuration, which generate $\sim$25K total forwarding rules. The configuration changes include adding/removing subnets, access control lists (ACLs), OSPF routes, etc. Figure 1 depicts the size of each configuration change and the corresponding number of changes in forwarding rules and ACL rules. On average, 228 lines of configuration ($\sim$0.3%) are changed, causing 146 forwarding rules ($\sim$0.6%) and 34 ACL rules ($\sim$0.9%) to change. Except for two updates, all configuration changes result in $<$600 changes in forwarding rules.

External route changes. Prior studies have shown that autonomous systems (ASes) may experience a high rate of BGP updates—e.g., four tier-1 ISPs each experience a median of $\sim$100K route updates per day [12]—yet only 1% (10%) of next-hops in the Internet change each day (month) [10]. Con-

sequently, we expect external route changes to have a small impact on a network's control and data planes.

To validate this hypothesis, we analyze 1 year of hourly RIB snapshots from a national R&E network [3]. We exclude hours in which a configuration change was made to ensure we only capture RIB changes caused by external route changes. We find <15% of hours have at least one RIB entry change, and <4% of hours have more than 10 RIB entries change.

Link/router availability changes. Link/router failures—caused by software upgrades, hardware faults, etc.—are common: e.g., the CENIC Digital California and High Performance Research networks experience a median of 5.1 and 38.5 failures per link per year, respectively [42], and tens of geographically distributed data centers operated by a large online service provider experience a median of 18.5 link failures and 3 device failures per day [17]. However, Steffen et al.'s experiments with 90 ISP topologies showed that for a randomly chosen ingress node and destination prefix, two-thirds of link failures do not impact the forwarding path from the ingress to the destination [40]. In other words, we expect only a fraction of FIB entries to change when links/routers fail.

2.2 Identifying behaviors to (re-)verify is hard

Identifying how changes in the control plane impact end-to-end forwarding behavior is important for assessing whether the changes: (1) have the intended effect, and (2) have any undesirable side-effects.

For the former, it is easy to identify which behaviors to examine, because these behaviors are effectively the "design requirements" for the change. For example, operators may propose a change in filters to restrict access to a new subnet; the effectiveness of the change can be assessed by examining reachability between the restricted and new subnet(s).

In contrast, it is difficult to identify which behaviors should be examined to determine whether a change is "safe." Some behaviors may be obvious, because they relate directly to the change: e.g., to assess the safety of the proposed change in route filters, operators should examine reachability from non-restricted subnets to the new subnet. However, a change can also impact seemingly unrelated behaviors—e.g., reachability from non-restricted subnets to existing subnets—or behaviors outside of an operator's purview—e.g., changes in a single data center may affect how a WAN load balances traffic across multiple data centers. The latter can arise especially when different teams manage different aspects of a network, networks are merged (e.g., due to an acquisition), or operators with historical knowledge of the network leave an organization.

A simple way to ensure a change does not negatively impact seemingly unrelated behaviors is to examine all categories of end-to-end behaviors for all (pairs of) prefixes. However, this is prohibitively expensive [9], and likely results in lots of unnecessary computation—e.g., if access from a subnet was already restricted, then further restrictions in access are unlikely to impact reachability. Ideally, only behaviors that could potentially be impacted should be examined.

In summary, repeatedly analyzing full snapshots of the control plane is wasteful, and determining which end-to-end behaviors to analyze is hard.

3 Overview

This section overviews DNA, a modular network analysis framework which can incrementally compute the "differences" in forwarding behavior that arise from "differences" in the control plane. We will first present the framework of DNA, and use an example to show its workflow. After that, we discuss three challenges when realizing DNA.

3.1 The DNA workflow

DNA is a modular framework with three stages, where each stage consumes and produces some forms of *differences*.

- The first stage consumes control plane changes and simulates the control plane to generate differences in data plane state (i.e., insertions/deletions of forwarding rules).
- The second stage updates a data plane model to generate differences in forwarding graphs (i.e., insertions/deletions of packet equivalence classes on edges).
- The third stage identifies and checks relevant end-to-end forwarding behaviors to generate differences in end-to-end properties (e.g., changes in reachability).

We use an example to show how these three stages work.

An example network. Figure 2(a) shows an example network which is used throughout the paper. The network has five routers running BGP. Router E announces two /24 prefixes and one /16 prefix. There is an outbound route filter at port 2 of router E, which filters routes for the /16 prefix, and an inbound ACL at port 2 of router D, which drops all traffic to the two /24 prefixes. The green and yellow arrows represent the best routes selected at each router.

To simulate a change, we consider a single link failure, i.e., the link between router C and E fails. As a result, router A can no longer reach the two /24 networks. We show how DNA can uncover this change in reachability.

Stage 1. Differential control plane simulation (§4). This stage takes as input differences in control plane state, incrementally simulates the control plane, and outputs differences in data plane state. Differences in control plane state are insertions/deletions of configuration lines, external routes, or links/routers, and differences in data plane state are insertions/deletions of forwarding/ACL rules. Generally, simulating a control plane entails modeling the route propagation, filtering, and selection behaviors within and across distributed routing protocols to compute the converged control plane state and produce a concrete data plane [1, 7, 14, 34, 37, 38]. To be incremental, we must transition from one converged state to

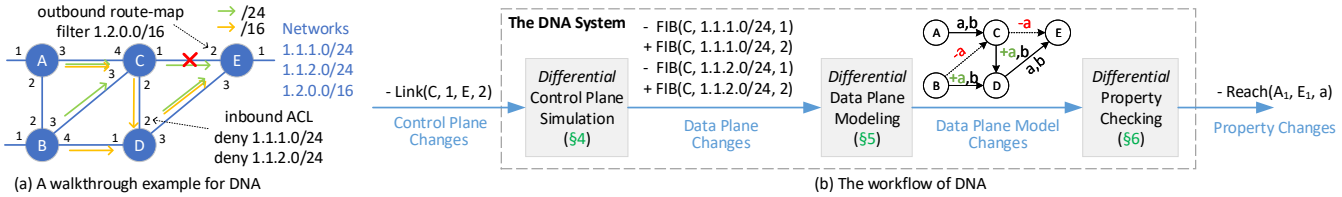


Figure 2: An example demonstrating the workflow of DNA.

another, accounting for the impact of control plane changes on route propagation, filtering, and/or selection across routers.

In our example, the link failure between port 1 of C and port 2 of E will be encoded as a deletion of a record $Link(C,1,E,2)$, which is the input the Stage 1. Given this input, DNA simulates the control plane and generates FIB differences: e.g., at router C, the two /24 routes whose output port is 1 (2) are deleted (inserted), as shown Figure 2.

Stage 2. Differential data plane modeling (§5). This stage takes as input differences in data plane state, incrementally constructs a data plane model, and outputs differences in the data plane model. Generally, a data plane model partitions the packet space into *Equivalence Classes (ECs)*, each of which represents a set of packets with the same forwarding behavior through the network [26]. Then, the model concisely encodes the forwarding behavior of packets with a forwarding graph, where each edge is labelled with ECs that can traverse it [18, 48], as shown in Figure 2. When rules are inserted or deleted, ECs are split, merged, and transferred among edges, to reflect the forwarding behavior change.

In our example, there are three ECs in total, where a represents $[1.1.1.0, 1.1.2.255]$, b represents $[1.2.0.0, 1.2.255.255]$, and c represents all other IP addresses (which are not shown here). Since forwarding rules for the two /24 prefixes which output to port 1 are deleted, and those which output to port 2 are inserted, EC a "transfers" from edge (C,E) to edge (C,D) and from edge (B,C) to edge (B,D), as shown in Figure 2. Thus, the differences of data plane model consist of two insertions and two deletions for EC a on these four edges.

Stage 3. Differential property checking (§6). This stage takes as input differences in the data plane model (i.e., insertions and deletions of ECs on the forwarding graph), incrementally computes relevant forwarding behaviors, and outputs the differences in properties. Here, the differences in properties, which we term as *differential properties*, are defined as insertions and deletions of forwarding properties, including reachability, waypointing, load balancing, etc. DNA computes differential properties by traversing the forwarding graphs from edge ports, which are ports that connected to the hosts, servers, or other networks. The traversal starts with a set of all affected ECs, which are updated by intersecting with those ECs labelled on the edge. The traversal ends when the set of ECs becomes empty or another edge port is reached.

In our example, we have three edge ports, i.e., port 1 of A, B, and E. By traversing from these edge ports, we get the differential reachability as $-Reach(A_1,E_1,a)$, meaning that packets belonging to EC a can no longer reach port 1 of E from port 1 of A. As will be shown in §6.2, DNA optimizes the above computation by traversing directly from the *change points*, where network forwarding behaviors change (B and C in this example). Note that network invariants like loop-freedom and blackhole-freedom are covered by differential reachability, since loops or blackholes will result in some pairs of end points becoming unreachable.

3.2 Challenges in realizing DNA

Realizing each of the three stages of DNA requires addressing the following three challenges, respectively.

(1) How to achieve control plane simulation in a way that is easy to extend? Network control plane has complex semantics, and simulating a control plane to cover all relevant feature often lead to complex code base. Generally, an incremental algorithm can be much more complex than its non-incremental counterpart [39]. Thus, incremental control plane simulation can be difficult to build. In addition, this is not a once-for-all task, considering the semantics of the control plane are still evolving, and new vendor-specific features emerge. Therefore, DNA needs to realize incremental control plane simulation in an easy-to-extend way.

(2) How to efficiently update data plane model for batched rule updates? Existing data plane verifiers are designed to consume single-rule updates—i.e., for each rule insertion or deletion, they individually update the model. For DNA, however, the differences in data plane state consist of a batch of rule updates. As an example, a link failure causes deletions of rules for multiple prefixes which are previously forwarded through the failed link. In such a setting, consuming single-rule updates can result in redundant computation, since these rule updates are often correlated. Therefore, DNA needs to optimize the data plane model update algorithm for batched rule updates.

(3) How to determine and only re-check affected properties? Existing data plane verifiers can incrementally check invariants like blackhole-freedom and loop-freedom by only re-verifying the ECs affected by a FIB update. DNA, however, needs to check all properties of some type (e.g., all-pairs reachability), and re-checking all these properties is wasteful since often only a small portion of the properties are affected. Therefore, DNA needs a way to determine which properties

are affected, which is not easy: e.g., it is unclear which reachability properties may be affected by a change in OSPF cost.

We describe how we address these challenges in §4–§6.

4 Differential Control Plane Simulation

As the first stage, DNA maps differences in configurations, external routes, and available links/routers to differences in FIBs. In the following, we show how DNA achieves this mapping in an incremental and easy-to-extend manner.

4.1 Modeling the control plane

A network control plane computes routes in a recursive way: each router receives routes from neighboring routers, filters and/or modifies the routes, locally ranks the routes to select the best routes, and advertises the best routes (which may be filtered/modified) to neighboring routers. The steps are repeated until routing converges—i.e., no more changes are made to any routing information bases (RIBs).

Leveraging differential dataflow for automatic differential computation. The aforementioned process fits well into the *dataflow* programming model. A dataflow program corresponds to a directed graph where vertices represent operators (i.e., functions that transform data), and edges represent the flow of data between operators [20]. For control planes, filtering, modifying, and selecting routes can be modeled with operators, and the propagation of routes between RIBs corresponds to the flow of data.

Differential Dataflow (DD) [35] is one dataflow programming framework which supports general incremental computation for recursive dataflows. This is achieved with a set of differential operators like `join`, `count`, etc. which efficiently produce differences in outputs from differences in inputs.

The left of Figure 3 shows a part of the dataflow program for OSPF route propagation using operators offered by DD. It `joins` the routes stored in a collection `BestOSPFRoute` with another collection `OSPFNeighbor` to model the propagation of routes from one router to another router; `filters` the routes where the origin of the route is the router itself (in order to prevent loops); `joins` with the collection `OSPFCost` to get the OSPF cost configured on the interface where the route is received; `maps` the routes to new routes with the OSPF cost updated by adding the interface's cost; and finally `joins` with `InterfaceIP` to get the IP of the interface to produce routes `OSPFRoute`, which will be further be processed to produce the `BestOSPFRoute` (omitted here).

Leveraging differential Datalog for better extensibility. As can be seen in the above example, modeling route computation directly using low-level operators offered by DD is less intuitive and can take a lot of effort, making the model hard to extend to new (vendor-specific) protocols or features. Therefore, we leverage *Differential Datalog (DDlog)* [2], a Datalog programming language built on top of DD. In the following, we give some preliminary to Datalog, and introduce our DDlog-based control plane model.

A Datalog program consists of a set of *facts* and *rules*. A fact is a statement like "interface *intf* of router *X* has OSPF cost *cost*". This fact can be represented with $\texttt{OSPFCost}(X, intf, cost)$, where `OSPFCost` is termed a *relation*. A rule takes the form of $R_1(u_1) :- R_2(u_2), \ldots, R_n(u_n)$, where each R_i is a relation, meaning $R_1(u_1)$ holds if $R_2(u_2), \ldots, R_n(u_n)$ hold. Given some *base facts*, one can derive new facts by firing the Datalog rules.

DDlog-based control plane model. The right of Figure 3 shows the corresponding DDlog rule for the corresponding dataflow model on the left. As we can see, DDlog allows us to only focus on how routes (i.e., facts) are derived, without caring about the sequence to join, map, or filter routes. Additionally, unlike other Datalog languages [5, 21], DDlog offers several useful data structures, such as vectors, which make the modeling of route computation much easier.

Figure 4 shows the flow of data in our DDlog-based control plane model. If relations *A* and *B* appear on the left and right side of a rule, respectively, then there is an edge from *B* to *A* in the graph. If multiple relations appear on the right side of a rule to derive relation *A*, we merge their edges to *A*.

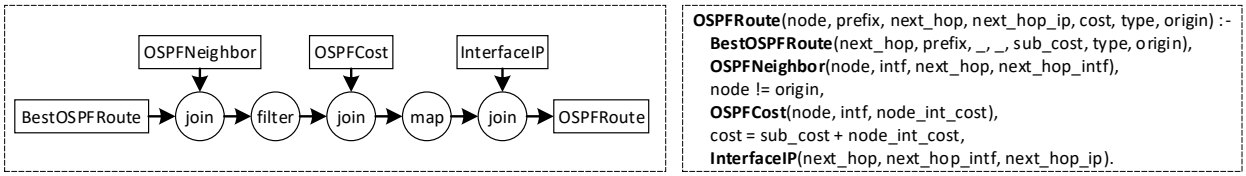
There are three types of relations: input relations, output relations, and intermediate relations. The input relations contain base facts including: (1) configurations for routing protocols, e.g., `BGPNet` contains the subnets imported to BGP; (2) network topology, e.g., `Link` contains L3 links; and (3) external routes, e.g., `ExtRoute` contains routes announced by ISPs that are out of scope of our model. There is a single output relation, i.e., `FIB`, and multiple intermediate relations, e.g., `GlobalRIB` which contain derived facts, e.g., routes.

The DDlog-based control plane model treats changes in the control plane as insertions and deletions of facts in input relations. For example, modifying the OSPF cost on interface 3 of router *D* from 10 to 100, is treated as two changes $-\texttt{OSPFCost}(D, 3, 10)$ and $+\texttt{OSPFCost}(D, 3, 100)$. The resultant changes in data plane state are insertions and deletions of facts in the output relation, i.e., `FIB`.

4.2 Executing the control plane model

The control plane model, which is a DDlog program, will be compiled into a DD program for execution. When executing the DD program, changes are propagated in the dataflow graph, and at each operator the changes in input will be mapped to changes in output. Since the dataflow is recursive, the propagation continues for multiple iterations, until there are no more changes (fixed point is reached). The insertions and deletions in the output relation `FIB` will be returned, and will be fed to Stage 2 (§5).

Figure 5 shows the execution of the control plane model for our example network. We have over-simplified the execution



(a) The dataflow graph (b) The Datalog program

Figure 3: Part of dataflow graph with the corresponding Datalog program snippet for OSPF route propagation.

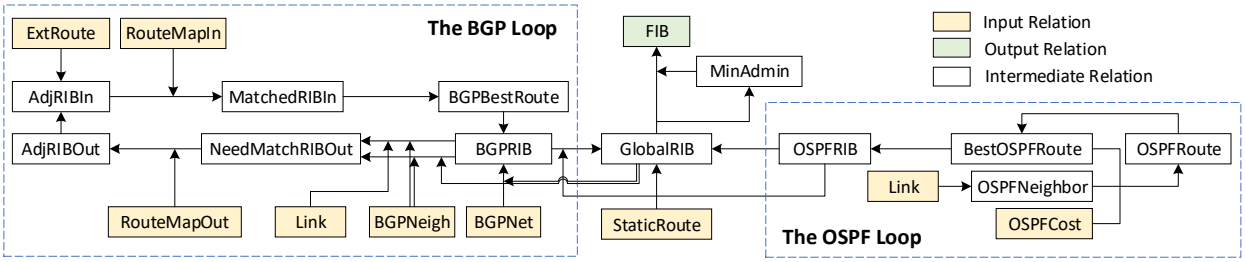


Figure 4: The flow of data in our control plane model. For simplicity, only the core relations are shown.

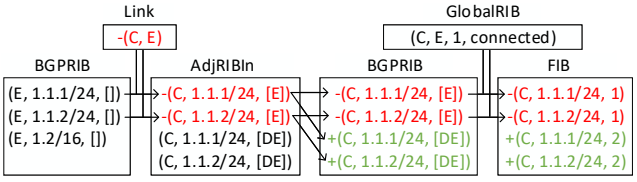


Figure 5: Control plane simulation for the example network.

by removing changes to a lot of intermediate relations and only focusing on changes to `Link`, `AdjRIBIn`, `BGPRIB`, and `FIB`. Here, the link failure $-(C,E)$ is input to the DD computation engine. The deletion will be `joined` with existing facts in `BGPRIB` to derive new facts. Since the change is a deletion so the changes it derives are also deletions. After multiple intermediate steps, the change derives $-(C, 1.1.1/24, [E])$ and $-(C, 1.1.2/24, [E])$ in `AdjRIBIn`, meaning deletions of the received routes for prefixes $1.1.1.0/24$ and $1.1.2.0/24$ with AS path $[E]$ at router C. These two deletions in `AdjRIBIn` will trigger deletions of the old best routes and insertions of the new best routes for router C in `BGPRIB`. The changes in `BGPRIB` will then be joined with `GlobalRIB` to generate four changes in the output relation `FIB`. For simplicity, only the join of two deletions in `BGPRIB` and the corresponding fact in `GlobalRIB` are shown in Figure 5.

4.3 Optimizations

Customizing functions for efficiency. As noted above, using operators like `join` and `map` in DD can express simple operations like route propagation (routes are sent to neighboring routers, costs are updated, etc.). However, directly modeling more complex operations—e.g., BGP best route selection, applying route policies, etc.—using these DD operators requires a lot of operators, making the evaluation less efficient.

For example, suppose we select best routes from received routes R based on two conditions: local preference (LocPref) and path length (PathLen). Using DD, we need to group routes in R by prefix and use the aggregation function `max` to compute the highest LocPref value for each prefix, and then `join` the resultant collection $R1$ with R to obtain another collection $R2$ which contains routes with the highest LocPref. Similarly, we need to compute another two collections for PathLen. Correspondingly, if we use DDlog, we need to declare two rules and two relations for each condition. The original Datalog-based version of Batfish [14] used the above approach to realize BGP best route selection. Since there are many criteria for BGP best route selection (Cisco uses 13 criteria), we need a lot of DD operators, or correspondingly a lot of DDlog rules, making the model inefficient to evaluate.

To make the model efficient to evaluate, we realize complex operations (e.g., best route selection and route policies) with customized functions, which can be wrapped inside a `Reduce` operator offered by DD. Appendix C gives the code snippet of the function for best routes selection. In our experiments, by just customizing the process of best route selection, we can achieve a ∼40% speedup (§8.1).

Partitioning routes for parallel simulation. For the same routing protocol, the propagation of different routes (prefixes) are largely independent. In the absence of route aggregations, two BGP routes $1.1.1.0/24$ and $1.1.2.0/24$ propagate independently through the network. Therefore, we partition the routes of the same routing protocol into groups, for parallel evaluation, and merge their results to obtain the RIBs for this protocol. In the following, we discuss why this works when there are multiple protocols and route aggregations.

(1) Route Dependency. When there are multiple routing protocols, routes may have dependencies. For example, BGP routes may depend on the OSPF routes for the loopback inter-

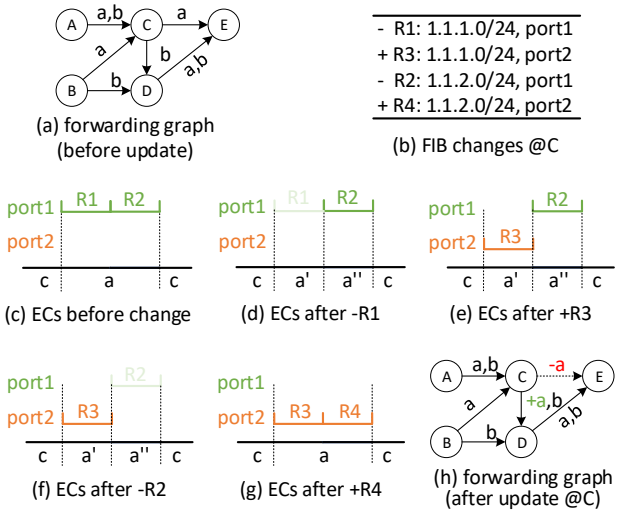


Figure 6: Data plane model update for the example network.

faces of all iBGP peers in the same AS. We adopt a simple approach where we group BGP routes and OSPF routes separately, and schedule BGP groups only after all OSPF groups are finished. More sophisticated scheduling [37] can be used to handle more complex dependencies.

(2) Route Aggregation. Sometimes, different routes may be correlated due to `route aggregation`. For example, a router can aggregate routes for 1.1.1.0/24 and 1.1.2.0/24 into a single route (1.1.0.0/16) when advertising to its neighbors. Although it may seem this correlation prevents routes for 1.1.1.0/24 and 1.1.2.0/24 from being computed separately, our method is not affected since each instance has all the rules and base facts describing route aggregation. Thus, even if these two routes are in different groups, both of them can be aggregated into 1.1.0.0/16. We can remove the duplicated routes of 1.1.0.0/16 when merging the routes of multiple instances.

5 Differential Data Plane Modeling

As the second stage, DNA maps differences in data plane state to differences in the data plane model. We accomplish this using APKeep [48], a state-of-the-art data plane model.

In the following, we first show that updating the model by treating each data plane update separately can result in redundant computation, and then show how DNA can leverage correlation among rule updates to reduce such redundancy.

5.1 Single-rule model update

We return to the example network to show how APKeep updates the data plane model. We only consider the rule updates at Router C, as shown in Figure 6(b). The rule updates at Router B are quite similar and thus not discussed here.

Step 1. Identifying forwarding behavior changes. For each rule update, APKeep identifies the packets that change forwarding behavior[1] by analyzing rule dependency. Returning to the example, after removing $R1$, APKeep determines that packets which previously match destination IP addresses in $1.1.1.0/24$ will not match any lower-priority rule, and thus will be dropped. Then, the forwarding behavior change will be a 3-tuple $(1.1.1.0/24, port1, drop)$ which specifies the affected packets, old, and new output port, respectively.

Step 2. Updating the forwarding graph. For each change, APKeep updates the ECs, and transfers the updated ECs on the forwarding graph. Specifically, APKeep iterates over all ECs assigned to the old port, and check whether each EC belongs to or intersects with the affected packets. For the former, the EC will be transferred directly; while for the latter, the EC needs to be split before the transfer. In this example, the affected packets $1.1.1.0/24$ will split EC a into two ECs, a' for $1.1.1.0/24$, and a'' for $1.1.2.0/24$, as shown in Figure 6(d). Then, a' will be transferred from port 1 to port $drop$ (a default port not shown here).

Figure 6(e) shows the insertion of $R3$, where step 1 identifies a behavior change $(1.1.1.0/24, drop, port2)$, and step 2 transfers a' from port $drop$ to port 2. Figure 6(f) shows the deletion of $R2$, and Figure 6(g) shows the insertion of $R4$, after which EC a' and EC a'' have the same forwarding behavior and are merged into a single EC a. Figure 6(h) shows the resulting differences of data plane model which are insertions/deletions of EC a on two edges.

In large networks, a configuration change may produce hundreds or thousands of rule updates, and performing the above two steps for each of them is slow. For example, failing a link in a fat tree with 180 routers results in over 3K rule updates. Even though a single rule update takes only 1ms, it still amounts to 3 seconds.

5.2 Batched model update

We observe that even when there are many rule updates, they are highly correlated, such that we can batch them to reduce redundant computation. In the following, we consider two types of correlations.

Correlation among rule insertions and deletions. Rule deletions are often accompanied by rule insertions for the same IP prefix. Many configuration changes like changing a BGP local preference or OSPF link cost will make the router change the best routes for some destination prefixes. Each change of best route would translate into a deletion of the old rule and an insertion of a new rule.

Based on the above correlation, we can *batch rule deletions and insertions* for step 1. Returning to our example, deleting $R1$ and inserting $R3$ requires updating the data plane model twice. However, by batching the deletion of $R1$ and insertion of $R3$, we can directly identify the change

[1] In this section "forwarding behavior" refers to hop-by-hop forwarding, not end-to-end forwarding.

as $(1.1.1.0/24, port1, port2)$. Therefore, we only need to run step 1 once. Moreover, step 1 will be more efficient since we do not need to analyze rule dependency. Similarly, by batching the deletion of $R2$ and insertion of $R4$, we can use another run of step 1 to identify a change $(1.1.2.0/24, port1, port2)$.

Correlation among rule updates on the same device. Rule updates on the same device are often quite similar, e.g., route deletions/insertions have the same output port. Many configuration changes like bringing down/up an interface will delete routes that output to the interface, and add new routes that output to other interfaces.

Based on the above correlation, we can *batch forwarding behavior changes on the same device* for step 2. Returning to the example, the two changes $(1.1.1.0/24, port1, port2)$ and $(1.1.2.0/24, port1, port2)$ have the same old port and the same new port. Instead of performing step 2 for each of these two changes, we can batch them as a single one $(1.1.1.0/24 \vee 1.1.2.0/24, port1, port2)$, and run step 2 only once. Moreover, we can directly transfer EC a from $port1$ to $port2$, without splitting a, further reducing computation overhead. Since step 2 needs to check all ECs of the old port, each involving a BDD operation, it dominates the overall running time of model update, and by batching changes for step 2, we can significantly reduce the overall running time.

In sum, APKeep needs to run the above two steps for each of the four rule updates, while after batching DNA only needs to run step 1 twice (without analyzing rule dependency), and step 2 once (without splitting and merging of ECs). As a result, DNA can directly update the model as Figure 6(g), avoiding intermediate steps shown in Figure 6(d)-(f).

Note that some of the batching methods can also be applied to other data plane verifiers. For example, Delta-net [18] can also be modified to leverage the first correlation. However, it is not clear how Delta-net can leverage the second correlation.

6 Differential Property Checking

This stage tracks network properties and returns differences, which we call *differential properties*. We focus on differential reachability, differential waypointing, and differential load balancing. In this section, we define these differential properties, and introduce an algorithm to efficiently compute them.

6.1 Defining differential properties

A network can be viewed as a big switch providing connectivity among entities including hosts, servers, middleboxes, external networks, etc. We term the ports at which these entities connect to the network as *edge ports*. We are interested in analyzing forwarding properties between the edge ports.

A forwarding property is defined in terms of a pair of edge ports (e_s, e_d), an equivalence class (ec), and other property-specific parameters. We focus on three types of properties:

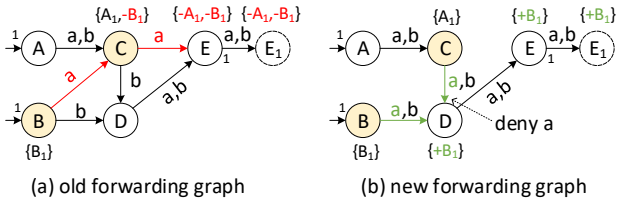


Figure 7: Differential reachability computation for the example network.

- $Reach(e_s, e_d, ec)$—packets in ec can reach e_d from e_s.
- $Waypoint(e_s, e_d, ec, w)$—packets in ec can reach e_d from e_s, traversing waypoint w.
- $LoadBalance(e_s, e_d, ec, n)$—packets in ec can reach e_d from e_s and are load balanced among n forwarding paths.

Other properties like isolation, bounded path length, etc. [4, 6], can be similarly defined. $Properties_c$ denotes the set of properties control plane c satisfies.

Given two control planes c_1 and c_2, the differences in properties are defined as $\Delta Properties_{c_1 \to c_2} := Properties_{c_2} - Properties_{c_1}$. $\Delta Properties_{c_1 \to c_2}$ is a multiset, where each item can have multiplicity $+1$ or -1. In the following, we consider the change of configuration $c_1 \to c_2$, and omit the subscripts. For example, $\Delta Properties = \{-Reach(A_1, E_1, 1.2/16)\}$, then we know this prefix is previously reachable from A_1 to E_1, but becomes unreachable after the change. This is perhaps what the operators desire if they want to prevent A_1 from reaching the prefix at E_1, or it can be a violation of operator intent if it is unexpected. As another example, $\Delta Properties = \{-Waypoint(A_1, E_1, 1.2/16, C)\}$ means this prefix will no longer traverse the waypoint C, which may violate security policies. Finally, $\Delta Properties = \{-LoadBalance(B_1, E_1, 1.2/16, 2), +LoadBalance(B_1, E_1, 1.2/16, 1)\}$ means the number of (disjoint) paths between B_1 and E_1 decreases from 2 to 1, which may cause congestion. By looking at differences in properties, instead of compliance of (all or user-specified) properties, operators can better understand the impact of prospective changes to their networks.

6.2 Computing differential properties

In this section, we show how DNA incrementally computes differential properties. We use differential reachability as an example, and discuss how to extend to the other two properties. The left of Figure 7 shows the forwarding graph of the running example. There are three edge ports, and here we only show the reachability from port 1 of A and B (denoted as A_1 and B_1), to port 1 of E (denoted as E_1).

A straightforward way to compute differential reachability is to compute the reachability for the old forwarding graph (before change) and the new forwarding graph (after change), and compute the difference. The reachability for the old graph is computed as follows. First, we start from each edge port with all ECs. Then, at each node, we com-

pute the conjunction of these ECs with the ECs marked on the edges, and move to the next hop with the conjunction of ECs. The traversal stops until another edge port is reached. For example, starting from A_1, ECs a and b can reach E_1, therefore we have a reachability $Reach(A_1, E_1, \{a, b\})$; similarly, we also have $Reach(B_1, E_1, \{a, b\})$. The reachability for the new graph is computed in a similar way, resulting in $Reach(A_1, E_1, \{b\})$ and $Reach(B_1, E_1, \{a, b\})$. Therefore $\Delta Properties_{c_1 \to c_2} = \{-Reach(A_1, E_1, a)\}$. We name this straightforward method as TraverseAll.

According to our experiment, the TraverseAll method can take tens to thousands of seconds to compute the differential rechability. To reduce the running time, we apply the following two optimizations.

(1) Only traversing with ECs whose forwarding behaviors are affected. In this example, we only need to start the traversal with EC a, since EC b is not affected. This optimization has been used by exiting realtime data plane verifiers, which check loops or blackholes by only traversing with those affected ECs. We name this method as TraverseAll-Inc. However, even there are only few ECs affected, TraverseAll-Inc still needs to enumerate all the pairs of edge ports, which can still take a long time if the network is large.

(2) Directly traversing from change points instead of from all edge ports. Here change points refer to nodes whose forwarding behaviors change (B and C in the example). Since the traversals before the change points are not affected, it is not necessary to start the traversal from each edge port, e.g., in this example we can start the traversal from B and C with EC a. To make this optimization work, we need to incrementally maintain intermediate state recording the traversal before the change points, that is, for each node, which edge ports can reach this node. For example, on the old forwarding graph, node C should know that EC a can reach C from A_1 and B_1, such that when the traversal from C reaches E_1, we can know the reachability from A_1 and B_1 to E_1.

DNA enables both these optimizations, where optimization (2) is enabled as follows. For each *node* and *ec*, DNA maintains a set $EdgeSet(ec, node)$, which stores all edge ports from which *ec* can reach *node*. In this example the content of $EdgeSet(a, node)$ is marked aside *node* on the forwarding graphs. When traversing, we need to update $EdgeSet$ according to the rule that if an $n_1 \in EdgeSet(ec, n_2)$, and *ec* can reach n_3 from n_2, then we have $n_1 \in EdgeSet(ec, n_3)$. In this example, when traversing from C to E on the old forwarding graph, since $A_1, B_1 \in EdgeSet(a, C)$, we can derive $A_1, B_1 \in EdgeSet(a, E)$. Since we are traversing the old graph, A_1, B_1 should be deleted from $EdgeSet(a, E)$, as shown on the right of Figure 7. On the contrary, when traversing the new graph, the derived entries should be inserted into $EdgeSet$. Interested reader can refer to Appendix A for the algorithm to compute differential reachability.

Computing differential waypointing and load balancing. Unlike reachability, computing differential waypointing and load balancing requires tracking the forwarding paths between edge ports. Therefore, instead of maintaining the edge ports from which *ec* can reach *node*, $EdgeSet$ should maintain the forwarding path taken by *ec* before reaching *node*. When traversing, we need to update $EdgeSet$ according to the rule that if $p_1 \in EdgeSet(ec, n_1)$, and *ec* can reach n_2 from n_1, then we have $p_1 || n_1 \in EdgeSet(ec, n_2)$, where p_1 is a forwarding path, and $p_1 || n_1$ appends n_1 to p_1. For the running example, (B_1, B, C, E) will be deleted from $EdgeSet(a, E_1)$, and (B_1, B, D, E) will be inserted into $Edge(a, E_1)$ after the change. Suppose C is a waypoint, then the change in $EdgeSet(a, E_1)$ indicates that packets belonging to EC a, sent from B_1 will no longer traverse the waypoint C.

Computing properties under link failures. We can leverage differential property to compute properties under link failures. For example, we can compute reachability properties that hold when any single link can fail. First, we compute a set R of all reachability properties when no links fail, Then, we fail each link one by one, and after each failure we compute differential reachability. For each deletion of reachability property, we remove it from the set R. After failing each single link, R contains all reachability properties that hold under any single link failure.

7 Implementation

We implement DNA in Java. First, we use Batfish [1] to parse the configuration files into vendor-neutral configuration objects, and write a parser to generate a set of insertions of base facts for the DDlog program.

For stage 1, we model the control plane with 800 LOC in DDlog. The model currently supports BGP, OSPF, static routes, routing policies, redistribution, reflector, communities, etc. The control plane model is compiled by the DDlog compiler into a DD program for execution. For stage 2, we extend APKeep [48] to optimize the model update algorithm for batched rule updates. For stage 3, we implement an algorithm to compute differential reachability (Appendix A).

Additionally, we implement a scheduler in Python to parallelize the data plane generation. The scheduler uses the DDlog's CLI, and maintains multiple instances of the DDlog program, each of which is responsible for a group of prefixes.

8 Experiments

We evaluate DNA with both real and synthetic updates. We are interested in the following questions: (1) can DNA speed up differential property checking (§8.1 and §8.2)? (2) is incrementally simulating the control plane always better when the changes are large, and can parallelization help DNA better scale to large changes (§8.3)? (3) can DNA also speed up property checking under link failures (§8.4)?

Table 1: Types of synthesized changes.

ID	Update	Explanation
1	InterfaceUp	Bring up an interface
2	InterfaceDown	Shut down an interface
3	NetworkAdd	Add a subnet to advertise to BGP
4	NetworkDel	Delete a subnet to advertise to BGP
5	NeighborAdd	Add a BGP neighbor
6	NeighborDel	Delete a BGP neighbor
7	LocalPref	Change the local preference
8	MultiPath	Allow to select up to k paths
9	Aggregation	Add an aggregation rule
10	StaticRoute	Add a static route

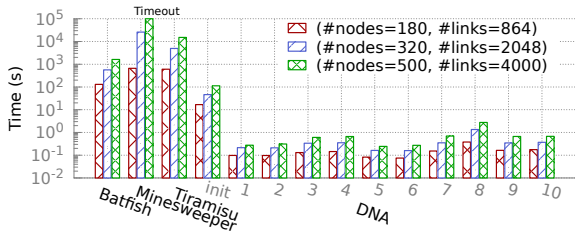


Figure 8: The time for DNA to compute differential reachability for synthetic changes on fat trees.

Setup. We run all the experiments on a server with two 12-core Intel Xeon CPUs @ 2.3GHz and 256G memory. Unless otherwise specified, a single core is used for these methods (except Batfish which is multi-threaded).

8.1 Synthetic changes

First, we evaluate the running time of DNA with synthetic changes. Specifically, we use different sizes of fat trees running BGP, where each node is assigned a distinct AS number and peers with all its adjacent nodes. We synthesize 10 different types of change, as shown in Table 1. Updates (1) and (2) can be used to simulate link failures and recovery, respectively. Updates (3) and (4) can be used to simulate changes in external routes. Update (7) adds a route map to change the local preference for routes received at one interface from 100 to 150 (more preferred).

Figure 8 reports the running time for DNA to compute differential reachability. For comparison, we also include the results for computing all-pair reachability using Batfish, Minesweeper, and Tiramisu. These tools can then compute differences of the all-pair reachability afterwards (the time to compute difference is not counted here). As we can see, DNA achieves a second-level running time for each control plane update, which is at least 3 orders of magnitude faster than existing tools—Minesweeper's [6] and Tiramisu's [4] main bottleneck is the number of links and end-host pairs, respectively. Here, init corresponds to the time for DNA to initialize, i.e., taking the original configuration snapshot as input, and running the three stages in the same way as processing a configuration update.

Figure 9 shows a breakdown of running time for the three stages, on fat tree (#node=500, #links=4000). As shown in Figure 9(a), DNA's control plane simulation takes less than 1 second for all updates except Update 8, while Batfish, which is not incremental, always takes 24 seconds. We also compare against a version of DNA without customized functions for best route selection (§4) (DNA$^-$), and observe the simulation is $\sim$40% faster with our customized functions. Although the absolute savings for a large fat tree is only $\sim$100ms, the difference is substantial when we consider link failures: e.g., control plane simulations for all single link failures (not shown) take $\sim$6 minutes longer with DNA$^-$.

As shown in Figure 9(b), directly running APKeep can take as long as 0.2 seconds, while DNA can achieve running time mostly less than 0.01 seconds. In most types of changes, DNA is 10$\times$ faster than directly running APKeep.

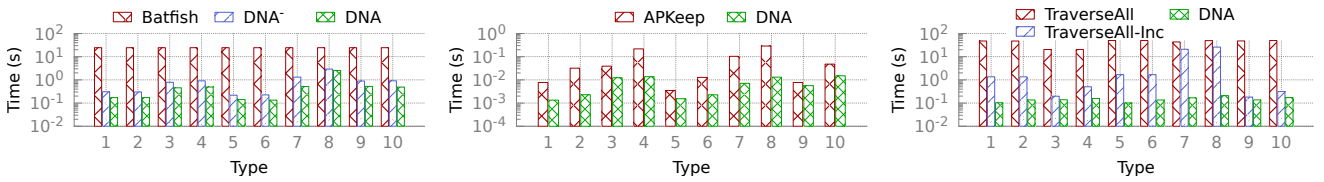
As shown in Figure 9(c), the TraverseAll method (traversing from all edge ports with all ECs) can takes as long as 400 seconds to recompute the all reachability properties. For TraverseAll-Inc (Traverse from all edge ports with only affected ECs) method runs mostly around 1 seconds, but can take more than 10 seconds for update 7 and 8. The reason is that in these two updates, the affected ECs appear at all edge ports, and TraverseAll-Inc still needs to traverse from all edge ports. In contrast, DNA takes around 0.1 seconds, a speedup of 1-2 orders of magnitude compared to TraverseAll-Inc. For Updates 9 and 10, DNA and TraverseAll-Inc take roughly the same small amount of time. The reason is that there are a small number of affected ECs, but a large number of change points. For example, adding a static route only affects ECs overlapping with the route. However, since the route will be advertised by BGP, the ECs will change forwarding behavior at all nodes in the network. Therefore, both DNA and TraverseAll-Inc need to traverse from all edge ports.

We also experiment on fat trees running OSPF, and the trend is similar. The results can be found in Appendix §B.

8.2 Real changes

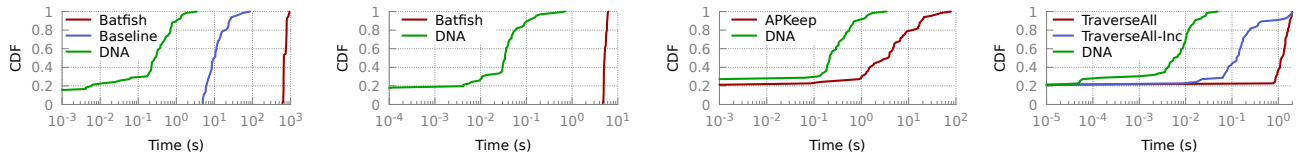
In addition to synthesized change, we also experiment with a real trace of configuration changes collected from the backbone network of a university campus. In total, the network consists of 28 routers and 50 physical links, running OSPF. The trace consists of 67 configuration snapshots spanning over three months. We compute the differences among consecutive snapshots to create 66 updates, which are fed to DNA for verification. The statistics on the network updates has already been shown in Figure 1.

Figure 10 shows the running time for the three stages of DNA. We compare the overall running time with Batfish, since Minesweeper times out (>1h per update). Also, we include the results for Baseline, which uses Batfish to generate



(a) Stage 1. Control Plane Simulation (b) Stage 2. Data Plane Modeling (c) Stage 3. Property Checking

Figure 9: The breakdown of running time for synthetic changes on fat tree (#nodes=500, #links=4000).



(a) Overall running time (b) Control Plane Simulation (c) Data Plane Modeling (d) Property Checking

Figure 10: The time to verify configuration changes of campus network.

the new data plane, APKeep to update the data plane model, and Traverse-All to compute differential reachability. We can see that DNA takes <1 second per update for 90% of all updates, while Batfish takes >500 seconds per update. Baseline is faster than Batfish by using realtime verifier (i.e., APKeep), but still takes tens of seconds per update on average.

Also note that for ∼20% of updates, the running time of DNA is less than 10ms. The reason is that for these updates, some interfaces or ACL rules are added without taking effect, and the output of the first stage is empty. The second and third stage are not even invoked. However, it is hard and risky to manually determine whether a change in configuration files has effect on the network, and without DNA we still need to check them using tools like Batfish or Minesweeper.

For control plane simulation, we compare the results of DNA with those of Batfish. For more than 90% of changes, the generation time is less than 0.12 seconds. While Batfish takes more than 4 seconds for each single change. This shows that incrementally generating data plane state is much faster than from scratch in real networks. For model update, DNA takes less than 1 second for 90% of changes, 10× faster than APKeep. For reachability verification, DNA takes strictly less than 0.1 second, while traversing from all edge points with all or affected ECs can take several seconds.

Compared to the synthesized changes on fat trees, where stage 1 dominates the overall running time, here stage 2 dominates the overall running time. The reason is that the largest fat tree (500 nodes, running BGP) has only 5K ECs, while the campus network has 45K ECs, due to the existence of ACLs.

8.3 Large changes and parallel simulation

In the previous two experiments, we mainly focus on small configuration changes. However, a network can occasionally experience large-scale changes [27], which may affect a large number of devices. We simulate large changes by shutting down a large number of interfaces in the campus network.

Figure 11 reports the time for DNA to incrementally simulate the control plane when failing a different number of links. For comparison, we also include the results of Batfish. As we can see, when failing a small number of links (say <10), incrementally simulating the control plane is much faster; while when the number increases to over 25 (single core), the incremental simulation becomes even slower than generation from scratch. This means that incremental simulation outperforms from-scratch simulation as long as the change sizes are smaller than some threshold (in our case, 50%). While since most of real changes are small (§2.1), incremental simulation would mostly be a better choice.

We also note that parallelizing the control plane simulation can increase such a threshold. Specifically, incremental simulation with 24 cores has larger improvement for larger changes. Even all links in the network were disconnected, the incremental simulation time is still comparable with from-scratch generation using Batfish.

We further study the effect of parallelizing control plane simulation on different size fat trees. For both BGP and OSPF, we randomly fail one node as well as all links connected to this node. Figure 12 reports the control plane simulation time for DNA with different number of cores. We can see the simulation speed increases with the number of cores. Due to the overhead of parallel simulation, the speed-up is not significant when the total time is already small (e.g., <1 second).

8.4 Enumerating link failures

One verification task is to check reachability under any single link failure. Using Batfish we need to enumerate each link failure and Minesweeper relies on SMT solvers to search for a counterexample where a link failure breaks reachability. DNA can leverage the similarity between the no link failure and single link failure to enumerate all link failures efficiently.

In this experiment, we evaluate the time to check reachability policies with any single link failure, i.e., whether two

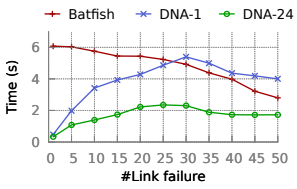


Figure 11: The running time for control plane simulation on the campus network.

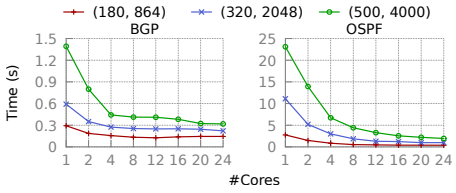


Figure 12: The running time for control plane simulation on fat trees, for one node failure with multiple cores.

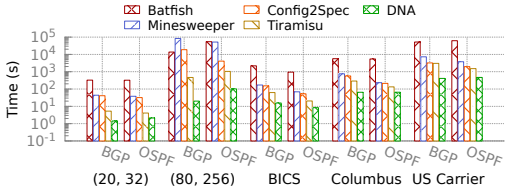


Figure 13: The total time for checking reachability under any single link failure. (n_1, n_2) represents fat tree with n_1 nodes and n_2 links.

hosts or ports are always reachable if any link can fail. We use two sizes of fat trees with 20 and 80 nodes, and three ISP topologies from Config2Spec. For each topology, we fail a link each time, and let DNA compute differential reachability.

For comparison, we run Batfish, Minesweeper, and Tiramisu to check the reachability between each pair of hosts on the same networks. We also include Config2Spec for comparison, since it is shown to outperform both Batfish and Minesweeper when checking all-pair reachability under link failures. Since Config2Spec also checks other properties like load balancing, we modify it to let it only compute all-pair reachability for a fair comparison.

As shown in Figure 13, DNA is at least 10× faster than Batfish and Minesweeper, 3× faster than Config2Spec, on all topologies. DNA is also faster than Tiramisu by 10× on the 80-node fat tree. The speedup of DNA on the three ISP topologies is not as remarkable as on fat trees. The reason is that the links on the ISP topologies are not as redundant as on fat trees, and a single link failure has a larger impact on reachability. The above results imply that by leveraging the similarity among network snapshots with and without a failed link, DNA can check reachability policies under any single link failure faster than existing control plane verifiers.

9 Related Work

Control plane simulation/emulation. Control plane verifiers have employed several approaches for simulating the control plane including: a Datalog engine [14], an explicit state model checker [37], a generalized variant of Dijkstra's algorithm [34], abstract interpretation [7], and custom simulation engines [1, 38, 45]. However, all these approaches restart the simulation from scratch when the control plane changes, and do not reuse any of the state from prior simulations. The preliminary version of this paper [47] introduces incremental network configuration verification, but does not parallelize incremental control plane simulation, and has limited support for incremental data plane modeling and property checking. Control plane emulators [32] can accommodate control plane changes in an incremental manner, but they scale poorly due to their use of actual routing protocol implementations.

Symbolic control plane verifiers. Symbolic control plane verifiers characterize the space of data planes the control plane may produce using graph algorithms [4, 15], SMT constraints [6, 43], or binary decision diagrams [13]. Even though some incremental graph algorithms exist [29] and SMT solvers offer some support for incremental solving [11], these capabilities are not sufficient to accommodate arbitrary control plane changes.

Data plane verifiers. Realtime data plane verifiers [18, 23, 26, 44, 48] can quickly analyze data plane changes (e.g., forwarding rule insertions), but cannot directly analyze configuration changes. DNA uses a realtime data plane verifier, APKeep [48], as one of its building blocks, and modifies APKeep to batch data plane changes for efficient processing, Other realtime data plane verifiers could also be modified to batch data plane changes and be used in DNA. NoD [33] uses Datalog, but NoD is not realtime.

Specification mining. Policy Units [8], Config2Spec [9], and Anime [25] infer a network's end-to-end behaviors from its configurations. We could compute differences in end-to-end behavior by applying such tools before and after a configuration change. However, due to the prevalence of small configuration changes (§2.1), such an approach unnecessarily duplicates computation in the same manner as applying a control plane verifier before and after changes (§1).

10 Conclusion

Differential Network Analysis (DNA) addresses a critical gap in control plane analysis: efficiently and effectively identifying differences in end-to-end forwarding behaviors arising from control plane changes. DNA uses a three-stage process that leverages advances in differential dataflow programming frameworks and data plane verifiers, along with domain-specific optimizations. Our evaluations using real and synthetic control plane changes show that DNA is able to compute differences in reachability in a few seconds—up to 3 orders of magnitude faster than state-of-the-art control plane verifiers. Thus, DNA provides a promising approach for operators to assess the impact of control plane changes.

Acknowledgements. We would like to thank the anonymous NSDI reviewers and our shepherd Brighten Godfrey for their valuable feedback. This work is partially supported by the National Natural Science Foundation of China (No. 61772412) and the National Science Foundation (No. 1763512).

References

[1] Batfish. https://github.com/batfish/batfish.

[2] Differential Datalog (DDlog). https://github.com/vmware/differential-datalog.

[3] Internet2 - visible backbone. https://vn.net.internet2.edu/Internet2/.

[4] A. Abhashkumar, A. Gember-Jacobson, and A. Akella. Tiramisu: Fast and general network verification. In *USENIX NSDI*, 2020.

[5] M. Aref, B. ten Cate, T. J. Green, B. Kimelfeld, D. Olteanu, E. Pasalic, T. L. Veldhuizen, and G. Washburn. Design and implementation of the LogicBlox system. In *ACM SIGMOD*, 2015.

[6] R. Beckett, A. Gupta, R. Mahajan, and D. Walker. A general approach to network configuration verification. In *ACM SIGCOMM*, 2017.

[7] R. Beckett, A. Gupta, R. Mahajan, and D. Walker. Abstract interpretation of distributed network control planes. In *ACM POPL*, 2020.

[8] T. Benson, A. Akella, and D. A. Maltz. Mining policies from enterprise network configuration. In *ACM IMC*, 2009.

[9] R. Birkner, D. Drachsler-Cohen, L. Vanbever, and M. Vechev. Config2Spec: Mining network specifications from network configurations. In *USENIX NSDI*, 2020.

[10] G. Comarela, G. Gürsun, and M. Crovella. Studying interdomain routing over long timescales. In *ACM IMC*, 2013.

[11] L. De Moura and N. Bjørner. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2008.

[12] A. Elmokashfi, A. Kvalbein, and C. Dovrolis. BGP churn evolution: A perspective from the core. *IEEE/ACM Transactions on Networking*, 20(2):571–584, 2012.

[13] S. K. Fayaz, T. Sharma, A. Fogel, R. Mahajan, T. Millstein, V. Sekar, and G. Varghese. Efficient network reachability analysis using a succinct control plane representation. In *USENIX OSDI*, 2016.

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

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

[16] A. Gember-Jacobson, W. Wu, X. Li, A. Akella, and R. Mahajan. Management plane analytics. In *ACM IMC*, 2015.

[17] P. Gill, N. Jain, and N. Nagappan. Understanding network failures in data centers: measurement, analysis, and implications. In *ACM SIGCOMM*, 2011.

[18] A. Horn, A. Kheradmand, and M. R. Prasad. Delta-net: Real-time network verification using atoms. In *USENIX NSDI*, 2017.

[19] K. Jayaraman, N. Bjørner, J. Padhye, A. Agrawal, A. Bhargava, P. C. Bissonnette, S. Foster, A. Helwer, M. Kasten, I. Lee, A. Namdhari, H. Niaz, A. Parkhi, H. Pinnamraju, A. Power, N. M. Raje, and P. Sharma. Validating datacenters at scale. In *ACM SIGCOMM*, 2019.

[20] W. M. Johnston, J. P. Hanna, and R. J. Millar. Advances in dataflow programming languages. *ACM Computing Surveys*, 36(1):1–34, 2004.

[21] H. Jordan, B. Scholz, and P. Subotić. Soufflé: On synthesis of program analyzers. In *International Conference on Computer Aided Verification*, 2016.

[22] S. K. R. Kakarla, A. Tang, R. Beckett, K. Jayaraman, T. D. Millstein, Y. Tamir, and G. Varghese. Finding network misconfigurations by automatic template inference. In *USENIX NSDI*, 2020.

[23] P. Kazemian, M. Chan, H. Zeng, G. Varghese, N. McKeown, and S. Whyte. Real time network policy checking using header space analysis. In *USENIX NSDI*, 2013.

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

[25] A. Kheradmand. Automatic inference of high-level network intents by mining forwarding patterns. In *ACM Symposium on SDN Research*, 2020.

[26] A. Khurshid, W. Zhou, M. Caesar, and P. Godfrey. VeriFlow: Verifying network-wide invariants in real time. In *USENIX NSDI*, 2013.

[27] H. Kim, T. Benson, A. Akella, and N. Feamster. The evolution of network configuration: a tale of two campuses. In *ACM IMC*, 2011.

[28] H. Kim, J. Reich, A. Gupta, M. Shahbaz, N. Feamster, and R. Clark. Kinetic: Verifiable dynamic network control. In *USENIX NSDI*, 2015.

[29] Y. Li, J. Jia, X. Hu, and J. Li. Real time control plane verification. In *Proceedings of the ACM SIGCOMM 2019 Workshop on Networking and Programming Languages*, pages 2–2, 2019.

[30] B. Liu, A. Kheradmand, M. Caesar, and P. B. Godfrey. Towards verified self-driving infrastructure. In *ACM HotNets*, 2020.

[31] H. H. Liu, X. Wu, W. Zhou, W. Chen, T. Wang, H. Xu, L. Zhou, Q. Ma, and M. Zhang. Automatic life cycle management of network configurations. In *SIGCOMM Workshop on Self-Driving Networks*, 2018.

[32] H. H. Liu, Y. Zhu, J. Padhye, J. Cao, S. Tallapragada, N. P. Lopes, A. Rybalchenko, G. Lu, and L. Yuan. Crystalnet: Faithfully emulating large production networks. In *ACM SOSP*, 2017.

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

[34] N. P. Lopes and A. Rybalchenko. Fast BGP simulation of large datacenters. In *International Conference on Verification, Model Checking, and Abstract Interpretation*, 2019.

[35] F. McSherry, D. G. Murray, R. Isaacs, and M. Isard. Differential dataflow. In *CIDR*, 2013.

[36] D. Plonka and A. J. Tack. An analysis of network configuration artifacts. In *Proceedings of the 23rd Large Installation System Administration Conference*, 2009.

[37] S. Prabhu, K.-Y. Chou, A. Kheradmand, P. Godfrey, and M. Caesar. Plankton: Scalable network configuration verification through model checking. In *USENIX NSDI*, 2020.

[38] B. Quoitin and S. Uhlig. Modeling the routing of an autonomous system with C-BGP. *IEEE Network*, 19(6):12–19, 2005.

[39] L. Ryzhyk and M. Budiu. Differential Datalog. In *International Workshop on the Resurgence of Datalog in Academia and Industry*, 2019.

[40] S. Steffen, T. Gehr, P. Tsankov, L. Vanbever, and M. Vechev. Probabilistic verification of network configurations. In *ACM SIGCOMM*, 2020.

[41] Y. E. Sung, X. Tie, S. H. Y. Wong, and H. Zeng. Robotron: Top-down network management at facebook scale. In *ACM SIGCOMM*, 2016.

[42] D. Turner, K. Levchenko, A. C. Snoeren, and S. Savage. California fault lines: understanding the causes and impact of network failures. In *ACM SIGCOMM*, 2010.

[43] K. Weitz, D. Woos, E. Torlak, M. D. Ernst, A. Krishnamurthy, and Z. Tatlock. Scalable verification of border gateway protocol configurations with an SMT solver. In *ACM OOPSLA*, 2016.

[44] H. Yang and S. S. Lam. Real-time verification of network properties using atomic predicates. In *IEEE ICNP*, 2013.

[45] F. Ye, D. Yu, E. Zhai, H. H. Liu, B. Tian, Q. Ye, C. Wang, X. Wu, T. Guo, C. Jin, et al. Accuracy, scalability, coverage: A practical configuration verifier on a global wan. In *ACM SIGCOMM*, 2020.

[46] H. Zeng, P. Kazemian, G. Varghese, and N. McKeown. Automatic test packet generation. In *ACM CoNEXT*, 2012.

[47] P. Zhang, Y. Huang, A. Gember-Jacobson, W. Shi, X. Liu, H. Yang, and Z. Zuo. Incremental network configuration verification. In *ACM HotNets*, 2020.

[48] P. Zhang, X. Liu, H. Yang, N. Kang, Z. Gu, and H. Li. APKeep: Realtime verification for real networks. In *USENIX NSDI*, 2020.

A An Algorithm for Computing Differential Reachability

Algorithm 1 summarizes how DNA traverses the new forwarding graph while avoiding redundant traversal. The traversal for the old forwarding graph is the same except Lines 6, 7, and 10. For ease of notation, we represent each edge port as a special node termed *edge node*. We traverse from each change point loc with all the affected ECs $pkts$ (Lines 2-3). Suppose we are traversing from v to w, then we generate $pkts'$ by intersecting $pkts$ with the ECs that can be forwarded to w according to the data plane model (Line 15), and record this information in R (Line 16). $(loc, w, pkts) \in R$ if $pkts$ can reach w from the change point loc. If w is another change point, where the affected ECs are $pkts''$, the traversal continues to w with $pkts' \backslash pkts''$ (Line 17-18). In this sense, we *delegate* the traversal of common ECs $pkts' \cap pkts''$ to the traversal starting from w, therefore avoiding the redundant traversal of common ECs. If w is not a change point, the traversal continues to w with $pkts'$ (Line 19-20). The traversal ends when the set of ECs becomes empty (Line 12-13).

After all traversals finish, the algorithm iterates over all affected EC δ, and for each EC, extracts the forwarding paths of δ (Line 5). Then, for each link (loc, w) on the path, it updates $EdgeSet(\delta, w)$ according to topological order of w (Line 6-7). If w is an edge node, it updates the reachability matrix *Reach* (Line 8-10).

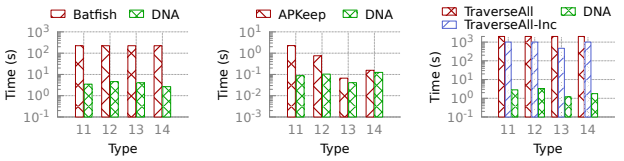
Algorithm 1: `DiffReach`(*Graph*, *Changes*)

Input: *Graph*: the forwarding graph; *Changes*: the set of data plane model changes.
Output: $Diff$: the differential reachability.

$R \leftarrow \{\}$;
foreach $(loc, pkts) \in Changes$ **do**
  `Traverse`$(loc, loc, pkts)$;
foreach $\delta \in \bigcup_{(loc,pkts)\in Changes} pkts$ **do**
  $Path(\delta) \leftarrow \{(loc, w) | (loc, w, pkts) \in R, \delta \in pkts\}$;
  foreach $(loc, w) \in Path(\delta)$ **do**
    $EdgeSet(\delta, w) \leftarrow EdgeSet(\delta, w) \cup EdgeSet(\delta, loc)$;
    if *w is an edge node* **then**
      foreach $e \in EdgeSet(\delta, loc)$ **do**
        $Diff \leftarrow Diff \cup \{+Reach(e, w, \delta)\}$;
Function `Traverse`$(loc, v, pkts)$**:**
  if $pkts = \emptyset$ **then**
    return;
  foreach $(v, w) \in Graph$ **do**
    $pkts' \leftarrow pkts \cap EC(v, w)$;
    $R \leftarrow R \cup \{(loc, w, pkts')\}$;
    if $(w, pkts'') \in Changes$ **then**
      `Traverse`$(loc, w, pkts' \setminus pkts'')$;
    else
      `Traverse`$(loc, w, pkts')$;

Table 2: Types of synthesized changes (OSPF). The IDs continue after Table 1.

ID	Update	Explanation
11	InterfaceUp	Bring up an interface
12	InterfaceDown	Shutdown an interface
13	LinkCost	Change the cost of one link
14	MultiPath	Allow to select up to k paths



(a) Stage 1. Control Plane Simulation (b) Stage 2. Data Plane Modeling (c) Stage 3. Property Checking

Figure 14: The breakdown of running time for synthetic changes on fat tree (#nodes=500, #links=4000).

B Experiments for Fat Tree Running OSPF

Figure 14 shows the breakdown of running time for DNA, on fat tree running OSPF. The updates are described in Table 2.

C Customized function for BGP best route selection in DDlog-based model

The following shows the code snippet of the function for best routes selection (simplified for ease of presentation).

```
BestRoute(route.node, route.dst, route.nexthop, ...) :-
  MatchedRIBIn[route1],
  var node = route1.node,
  var dst = route1.dst,
  var route = Aggregate((node, dst), select_best(route1))
      .

function select_best(g: Group<'K, AdjRIBIn>): AdjRIBIn {
  var route = group_first(g);
  for (route1 in g) {
    if (route1.LocalPref > route.LocalPref) route = route1
    else if (route1.LocalPref == route.LocalPref) {
      if (len(route1.path) < len(route.path)) route =
          route1
      // origin, MED, eBGP < iBGP, router-id, ...
  }};
  route // return the best route
}
```