NV: An Intermediate Language for Verification of Network Control Planes

Nick Giannarakis
Princeton University
USA
nick.giannarakis@princeton.edu

Devon Loehr
Princeton University
USA
dloehr@princeton.edu

Ryan Beckett
Microsoft Research
USA
ryan.beckett@microsoft.com

David Walker
Princeton University
USA
dpw@cs.princeton.edu

Abstract

Network misconfiguration has caused a raft of high-profile outages over the past decade, spurring researchers to develop a variety of network analysis and verification tools. Unfortunately, developing and maintaining such tools is an enormous challenge due to the complexity of network configuration languages. Inspired by work on *intermediate languages for verification* such as Boogie and Why3, we develop *NV*, an intermediate language for verification of network control planes. NV carefully walks the line between expressiveness and tractability, making it possible to build models for a practical subset of real protocols and their configurations, and also facilitate rapid development of tools that outperform state-of-the-art simulators (seconds vs minutes) and verifiers (often 10x faster). Furthermore, we show that it is possible to develop novel analyses just by writing new NV programs. In particular, we implement a new fault-tolerance analysis that scales to far larger networks than existing tools.

CCS Concepts: • **Networks** → **Protocol testing and verification**; • **Theory of computation** → **Automated reasoning**; *Verification by model checking*.

Keywords: Network Verification, Network Simulation, Control Plane Analysis, Router Configuration Analysis, Intermediate Verification Language

ACM Reference Format:
Nick Giannarakis, Devon Loehr, Ryan Beckett, and David Walker. 2020. NV: An Intermediate Language for Verification of Network Control Planes. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '20), June 15–20, 2020, London, UK*. ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3385412.3386019

PLDI '20, June 15–20, 2020, London, UK

ACM ISBN 978-1-4503-7613-6/20/06…$15.00
https://doi.org/10.1145/3385412.3386019

1 Introduction

Following the explosive rise in the number of internet users and the emergence of cloud computing, networks have seen significant growth in their size and complexity. However, network operators have a hard time keeping up with this unprecedented growth: in the past few years, network misconfiguration incidents have caused outages whose effects range from popular websites like Facebook being inaccessible [42] to cloud services going offline [35, 41, 46], to airlines grounding their flights [23, 34, 37].

To help improve network reliability, researchers have developed numerous verification and testing tools over the last decade. One wave of tools included Anteater [33], HeaderSpace Analysis (HSA) [26], Veriflow [27], NetKAT [4, 18, 44], NoD [31] and Bayonet [19]. These tools were designed to analyze the network *data plane*, the component of a network responsible for forwarding (or blocking) user traffic from point A to point B. The tools developed can check both deterministic and probabilistic properties of how packets flow through the data plane, and scale to networks with thousands of devices and millions of packet-forwarding rules.

A second wave of tools has focused on analyzing the network *control plane*. The control plane is the component of a network that gathers information about available routes (paths) to destinations and chooses which routes to use. Once the control plane has chosen its preferred routes, it passes them off to the data plane, which implements them. From time to time, the control plane will receive new information about the available routes (*e.g.*, when failures occur), and when it does, it may compute new routes to a destination. Tools that analyze the control plane, such as C-BGP [40], rcc [15], Batfish [17], ARC [21], ERA [14], Bagpipe [49], MineSweeper [6], FastPlane [32], and ShapeShifter [8], will analyze the router configurations that specify how routes are to be chosen and will compute the outcome of those choices

(*i.e.*, compute the routes that will be used by the data plane). These tools can then answer questions such as whether the control plane will compute a route from A to B (*i.e.*, control plane reachability), or whether it will compute route from A to B regardless of which single link failure occurs (*i.e.*, control plane fault tolerance). However, control plane tools are usually less scalable than the data plane tools, particularly when it comes to checking properties like control plane fault tolerance, which may involve computing different routes from A to B for every possible failure scenario.

This paper focuses exclusively on control plane analysis tools and the difficulty of building such tools.[1] Much of this difficulty arises from the fact that router vendors such as Cisco and Juniper each have their own proprietary configuration languages that consist of an enormous number of ad hoc commands that are specific to a particular protocol, such as OSPF or BGP. To analyze a network control plane, one must first parse these varied configuration languages and interpret their semantics.

Fortunately, colleagues at Intentionet have built Batfish [17], a tool that parses these configurations and creates an intermediate representation (IR) on top of which researchers can develop their own analyses. Indeed, researchers from a number of different institutions have built verification and simulation tools using Batfish as a front end [6–8, 14, 20, 21].

While Batfish is a remarkably useful front end for managing configurations, its IR does not fundamentally change the abstractions present in the vendor languages it supports—it represents those surface-level abstractions directly. Doing so has the advantage of being expedient and straightforward, but causes Batfish to inherit some of the less desirable properties of the configuration languages themselves. In particular, the Batfish IR is quite verbose: A recent examination of the IR showed it contained 24 statements and more than 100 types of expressions. It has 19 ways just to modify fields of a routing message. Implementing configuration analyses like MineSweeper [6] requires understanding the semantics of all these statements and expressions, which is challenging. Moreover, Batfish continues to evolve, adding new vendor features, and with them new components to its IR. This makes it a challenge to maintain analysis tools. It is even harder to develop and maintain tools like ShapeShifter [8] that need to transform conventional routing protocols, as the Batfish IR does not provide the "building blocks" necessary to represent new, non-standard routing protocols.

A Low-Level IR for Network Verification. Building sophisticated static analysis tools is no easy feat; it requires familiarity with different domains such as programming language semantics, automated reasoning techniques and, in this case, networking. Fortunately, we can simplify the task by first parsing source languages into a surface-level IR (*i.e.* Batfish's IR), and then encoding the semantics of this IR into a lower-level IR designed specifically for verification.

Such an architecture separates key concerns and simplifies each aspect of the pipeline. Indeed, this methodology has been used successfully for general-purpose programming languages in systems such as Boogie [30], Why [16], and CIVL [43]. The key challenge comes in the design of the low-level intermediate language for verification: it should be *succint*, *expressive* enough to encode the semantics of the source language, and *tractable* enough to admit efficient analysis of its features.

A Functional Language for Modeling Routing Protocols. Our first contribution is the design of *NV*, a functional language for modeling routing protocols. NV uses *conventional*, *expressive* and *compositional* constructs with ordinary, broadly-understood semantics, such as integers, booleans, functions, and records. Additionally, it allows users to express unknowns, such as potential failures or actions of neighboring networks, in a general way, using *symbolic values*, and lets users specify network properties via *assertions*.

NV's design provides building blocks for modeling both standard routing protocols and variations thereon. The latter is quite useful as large cloud providers have been known to tweak standard protocols for internal use; accommodating these tweaks is easier in NV. In addition, these building blocks can be used to implement transformations of NV programs. Optimizations, such as partial evaluation, or transformations, such as message [8] or topology [7, 22] abstractions can be implemented as NV-to-NV transformations, independent of one another and of the back-end analysis used. Prior to this research, designing a network verification tool involved simultaneously interpreting the low-level commands from various vendors (Cisco, Juniper, Arista, Force10, *etc.*), optimizing their representation, and converting them to the appropriate structure (*e.g.*, SMT constraints) all at once.

The most difficult design challenge in NV was oriented around the definition of a dictionary (finite map) type. These dictionaries need to be expressive enough to encode the key-value stores and the sets processed by most routing protocols, yet simple enough to admit efficient encodings both as SMT formulae for use in symbolic verification techniques, and as multi-terminal binary decision diagrams (MTBDDs) for simulation-based analysis. Such efficient encodings are key to scaling NV to large networks. We chose a design that revolves around *total* maps, allowing us to efficiently represent concepts such as the set of all possible route announcements; limited the allowed index operations to constants and symbolic variables; and carefully defined the aggregation operations (*map*, *mapIte* and *combine*).

To show the design of NV is effective, and that it carefully walks the tight line between expressiveness and tractability, we develop a translation from a subset of the control plane components of Batfish's IR to NV. This translation covers the

[1] Henceforth, whenever we refer to a "network", the reader may assume we are referring to the control plane of the network unless stated otherwise.

same control plane features as MineSweeper [6] (excluding iBGP) and a superset of the features that tools like ARC [21] model. In particular, it includes encodings of eBGP, OSPF, static and connected routes as well as redistribution between these protocols. Our default eBGP encoding does not include the full path, instead abstracting the path as its length (or, optionally, as the set of traversed nodes), but does include communities, local preference, multi-exit discriminator and policy that modify these fields. Our OSPF encoding includes OSPF areas and weighted link costs. This paper focuses exclusively on the control plane, and hence data plane features of configurations such as Access Control Lists, or protocols such as iBGP, whose analysis requires examination of the data plane, are not explored here.

Efficient and Flexible Analysis Tools. Our second contribution involves the implementation of two important network analyses over NV programs.

The first is an efficient network simulator, which computes the routes of the network described by an NV program. The simulator's design is inspired by the ShapeShifter simulator [8], which uses BDDs. However, where ShapeShifter is hard-coded to simulate a couple of chosen protocol abstractions, our simulator efficiently simulates any NV program. Moreover, thanks to the close correspondence between NV and OCaml, NV functions may be compiled and natively executed rather than interpreted. We show that the NV simulator is on average an order of magnitude faster than simulators such as Batfish (seconds for NV vs minutes for Batfish), while also consuming less memory (2GB vs 16GB) and scaling to larger networks (over a 1000 routers).

The second tool is a verifier that operates via translation to SMT formula. While the NV simulator scales to larger networks, NV's SMT-based verifier is able to process symbolic values (such as a value representing all possible neighbor advertisements). This SMT-based verifier is inspired by MineSweeper [6], but, like our simulator, operates over any NV program. For networks that do simple, shortest-path routing, MineSweeper and NV's SMT encoding have comparable performance, but when more complex policy is implemented, MineSweeper is 10x times slower than NV, and eventually times out as the network grows.

Novel Network Analyses. The third contribution of this paper is to show how NV's expressiveness helps researchers formulate new analyses with minimal effort. By providing a *programming language* that allows users to express network models easily, one can directly construct non-standard models that correspond to new analyses. NV makes it easy to play with these non-standard models, rapidly prototyping one and then the next to see what does and does not work.

We present a novel fault tolerance analysis implemented directly as a simple NV-to-NV program transformation. Contrary to previous approaches to fault tolerance ([6, 17, 21, 22]), this analysis scales well, computes precise routes, does not impose restrictions on the policy or features used, and is exhaustive. It is also quite a bit more flexible than past analyses: One can consider any combination of node and link failures they feel is relevant. Since fault tolerance is one of the most important and difficult properties of networks to check, this analysis on its own, despite its simplicity, is quite a breakthrough. And it was originally prototyped in just a few minutes, rather than taking weeks or months of work, as past fault tolerance analyses have.

2 An Overview of NV

2.1 How Routing Works

To learn how to route traffic to different destinations, routers rely on a number of distributed *routing protocols*. In these protocols, routers exchange *routing messages* with their neighbors. These messages (often called "routes") contain information about a path through the network to a given destination. Protocols carry different information such as the path itself (BGP), its hop count (RIP) or physical distance (OSPF, ISIS), or some complex combination of attributes (IGRP, EIGRP).

A router can be configured to run one or more protocols, and each protocol can be configured via a different set of policy parameters. For instance an Internet service provider might tag routes in BGP to indicate "this route was learned from Verizon", and then drop the route when exporting it to AT&T to avoid carrying transit traffic for free.

For each destination, a router selects a single best route among the ones received from neighbors (after applying any import policies). Subsequently, it forwards this route to its neighbors, perhaps after transforming it (*e.g.*, by incrementing the path length) and applying any export policies. Eventually, this procedure reaches a *stable state*, where all routers have selected the best routes available to them from their neighbors. It is this stable state that is of interest, as operators want to ensure that it adheres to their specification. Such specifications usually relate to connectivity, traffic engineering, security or other business considerations.

2.2 An Example Configuration

The distributed nature of routing makes configuration — and the subsequent verification task — a challenging problem. But that's not the sole challenge in working with configurations. Figure 1 shows a configuration snippet for a single router in the Cisco IOS format; briefly, the first part (lines 1-3) describes physical connections with other devices, while the second part (lines 5-14) describes the protocols that the router is running (OSPF, Static, BGP) and their configurations. For example, line 12 tells OSPF to inject statically configured routes, such as the one configured on line 5, into its routing table. Finally, lines 16-23 define user route policies. Even in this small snippet, some of the challenges for formal analysis of such configuration languages quickly become apparent:

```
1  interface Ethernet0
2    ip address 172.16.0.0/31
3
4  ip route 192.168.1.0 255.255.255.0 192.168.2.0
5  bgp router 1
6    redistribute static
7    neighbor 172.16.0.1 remote-as 2
8    neighbor 172.16.0.1 route-map RMO out
9
10 router ospf 1
11  redistribute static metric 20 subnets
12  distance 70
13  network 192.168.42.0 0.0.0.255 area 0
14
15 ip community-list standard comm1 permit 1:2 1:3
16 ip prefix-list pfx permit 192.168.2.0/24
17 route-map RMO permit 10
18   match community comm1
19   match ip address prefix-list pfx
20   set local-preference 200
21 route-map RMO permit 20
22   set metric 90
```

Figure 1. A small fragment of a router configuration.

1. Complex instruction set: Many distinct commands perform logically similar operations. For example, line 13 assigns the value 70 to the administrative distance of redistributed routes, and line 21 assigns the value 200 to the BGP local-preference parameter. Similarly, they use specific data structures, such as `prefix-list` and `community-list` to match routes, instead of generic data structures that serve multiple purposes. Cisco IOS [10] alone contains over 15000 configuration commands, and over 300 for BGP alone.

2. Lack of reusable building blocks: Configuration languages lack *building blocks*. Building blocks are useful for creating new, non-standard structures, and expressing transformations or abstractions of protocols, which may accelerate verification. Programmable building blocks also allow engineers to describe "meta-protocols", such as our fault-tolerance analysis, and make it possible to easily attack old problems in new and better ways.

3. Incomplete semantics: The semantics of the protocols are not explicit in the configuration. Instead, they are scattered throughout the configuration and the protocol RFC. For instance, how the BGP protocol selects a route is partially captured through configuration and partially specified by the RFC. By making the semantics explicit in a well-understood metalanguage, we can open up the field of network reliability research to a broader audience.

NV is designed to address each of these concerns.

2.3 Encoding BGP in NV

The Border Gateway Protocol (BGP) is the protocol used to exchange routing information between networks on the internet. In its most basic form BGP implements shortest-path routing; however, the protocol includes several knobs which operators can use to implement more sophisticated policies. To define a model of BGP in NV one defines:

1. a *transfer* function, defining how routes are transformed as they are propagated through the network, and
2. a *merge* function, defining how a node selects a best route.

Figure 2a presents a cut-down model of BGP in NV. In this model, we consider a single destination and represent routes towards it as optional values. No value indicates the absence of a route; otherwise, a BGP route consists of a record with five components: the path length (`length`), an integer known as local preference (`lp`), an integer known as multi exit discriminator (`med`), a set of integers known as communities (`comms`), and an originator (`origin`) indicating which node initially announced the route. Intuitively, the local preference value allows an operator to override the shortest-path selection, while the multi-exit discriminator is used by external peers to provide a tie-breaker in case there are multiple connections to their network. Finally, communities are tags that a router can attach, remove or test on a route, allowing network operators to implement custom policies.

The *transfer* function computes the route to be propagated over an edge in the network using the route selected by the propagating node. In our working example it simply increases the path length by 1, but in general, it can be used to describe more complex transformations, such as the BGP policy of fig. 1. Finally, when choosing between two routes, the *merge* function selects the route with the highest local-preference value, or, if they are equal, the one with the shortest path length. If the path lengths are also equal the multi exit discriminator is used to break the tie.

2.4 Using NV for Verification

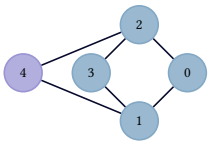
Having defined a model of BGP as in fig. 2a, we can use NV to verify properties about a network running BGP. To do so, we provide a topology, the initial route of each node and an assertion to verify as in fig. 2b.

Modeling Unknowns. Operators are often interested in verifying properties with respect to factors outside their control. Such factors could be potential hardware failures (a failed link or device), or a route sent from a peer network. Inspired from other solver-aided languages [47], NV uses *symbolic values* to model such "unknowns". A symbolic value is not bound to a single concrete value; rather it represents any possible value of its type. In the example of fig. 2b a symbolic value models the fact that node 4 may send arbitrary routes. Intuitively, this represents an external network (node 4) that peers with our network (nodes 0-3).

Specifying Properties. To specify a property of the converged (stable) state of the network we may define an *assertion*. An assertion is simply a predicate over a node and the final route it has selected. In the case of fig. 2b, our specification asserts that node 4 cannot hijack traffic from our network, *i.e.* nodes internal to our network should prefer the route that originated from node 0.

```
type bgp = {length:int; lp:int; med:int;
            comms:set[int]; origin:node}

type attribute = option[bgp]

let transBgp (e: edge) (x: attribute) =
  match x with
  | None -> None
  | Some b -> Some {b with length = b.length+1}

let isBetter x y =
  match x,y with
  | _, None -> true
  | None, _ -> false
  | Some b1, Some b2 ->
    if b1.lp > b2.lp then true
    else if b2.lp > b1.lp then false
    else if b1.length < b2.length then true
    else if b2.length < b1.length then false
    else if b1.med <= b2.med then true else false

let mergeBgp (u: node) (x y: attribute) =
  if isBetter x y then x else y
```

(a)




```
include bgp
let nodes = 5
let edges = {0n=1n;0n=2n;1n=4n;2n=4n;1n=3n;2n=3n}

symbolic route : attribute

let trans e x = transBgp e x

let merge u x y = mergeBgp u x y

let init (u : node) =
  match u with
  | 0n ->
    Some {length=0; lp=100; med=80; comms={}; origin=0n}
  | 4n -> route
  | _ -> None

let assert (u : node) (x : attribute) =
  match x with
  | None -> false
  | Some b -> if (u <> 4n) then b.origin = 0n else true
```

(b)

Figure 2. (a) shows a basic model of BGP in NV. (b) shows a network running BGP. Nodes 0-3 model an internal network and node 4 a peer announcing an unknown route to nodes 1-2. Can we verify that node 4 cannot hijack traffic from our network?

2.5 Implementing Network Analyses

To verify properties of a network, we need to compute its *stable states*, also known as its *solutions*. A *state* of a network is a labelling function $\mathcal{L}$ that maps each node to a route. Such a state is *stable* when given a node and the routes associated with its neighbors, there is no incentive for the node to adopt a route that differs from the one it already has. More precisely, we define the *choices* of node u as the routes received from neighbors, *i.e.* the set of routes computed by applying the transfer function over each edge (v, u) and the label $\mathcal{L}(v)$ of the neighbor:

$$\text{choices}(u) = \{a \mid e = \langle v, u\rangle,\ a = \text{trans}\ e\ \mathcal{L}(v)\}$$

The label of a node u can then be described as a combination of the *choices* provided from its neighbors and its initial route:

$$\mathcal{L}(u) = \text{init}(u) \oplus a_1 \ldots \oplus a_n$$
$$\textbf{when}\ \text{choices}(u) = \{a_1, \ldots, a_n\}$$
$$\textbf{and}\ x \oplus y \triangleq \text{merge}\ u\ x\ y$$

When the equation above holds for every node u in the network, the function $\mathcal{L}$ defines a stable state for the system. While not formulated in exactly the same way, these definitions reflect the same notion of stability as developed in earlier work by Griffin *et al.* [24] and Sobrinho [45].

Simulating Routing Protocols. One way to compute a stable state is through simulation. Network simulation mimics the route exchange process in which routers engage. It computes a fixpoint of the process in which each node uses the merge function to select the best route among the received ones, then modifies that route according to the transfer function and further propagates it to its neighbors (see section 5.1, algorithm 1). To achieve high performance, our simulator processes routes for multiple destinations in bulk [8], via a novel implementation of maps based on *multi-terminal binary decision diagrams* (MTBDDs) [11].

To simulate the network, we need to execute the merge and transfer functions. Typically network simulators do this by using an interpreter [17, 32]. However, interpreters can be slow, particularly when route policy is complex. An alternative (by two orders of magnitude in certain cases) relies on native execution to compute routes; this is enabled by NV's conventional language design. Section 5.1 details the compilation and linking with a simulator process.

SMT Verification. A second way to check properties of converged network states is through SMT [6]. The SMT-based approach does not model the convergence procedure; instead, it captures the stable solutions of the network using constraints. The challenge in creating an efficient SMT encoding is that high-level programs introduce abstractions that do not favor SMT reasoning. Much like a regular compiler, NV relies on an optimizing pipeline to produce tractable constraints. Section 6.2 shows that NV's systematic approach to optimizations results in improved performance compared to state-of-the-art control plane verifiers like MineSweeper.

The strength of the SMT-based analysis lies in its ability to perform symbolic reasoning. With respect to NV, this means that it can indeed reason about all possible assignments to a symbolic value. Returning to our working example in fig. 2b, the SMT analysis will refute our assertion: node 4 may send

```
include bgp
type attribute = option[(set[node],bgp)]

let trans e (x : attribute) =
  let (u,v) = e in
  match x with
  | None -> None
  | Some (s, b) ->
    (match transBgp e b with
     | None -> None
     | b' -> Some (s[u := true], b')

let merge u x y =
  match x,y with
  | _, None -> x
  | None, _ -> y
  | Some (s1,b1), Some (s2,b2) ->
    let b = mergeBgp u b1 b2 in
    if (b = b1) then (s1, b) else (s2, b)
```

Figure 3. Model capturing traversed nodes

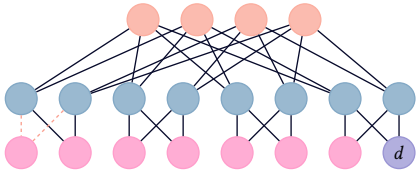


Figure 4. A FatTree network with two link failures inside a pod; the failures do not affect nodes outside the pod.

a better route than node 0 (*e.g.* one with same path length as the route from node 0 but lower `med` value) and since there are no configured route filters on nodes 1 and 2, node 4 can successfully hijack traffic from our network.

In contrast, exhaustive analysis using approaches based on normalization, such as simulation, is usually impractical. Instead, one explores a limited set of cases deemed interesting for the assertion under test, by by providing concrete values in place of symbolic ones.

2.6 Modeling Protocols

Large cloud providers are known to run modified versions of protocols in-house. For instance, a recent MineSweeper feature request was to change the way in which BGP ranks routes [38]. Such changes require familiarity with low-level code that generates the constraints encoding BGP's route ranking. Moreover, this change does not apply to other tools, such as Batfish, which would need another patch. In contrast, in NV it suffices to tweak the merge function (for example, by adapting the translation from Batfish to NV), and the change is automatically usable by all available analyses.

Furthermore, operators have information specific to their networks and the properties they want to check. They can leverage this information and NV's flexibility to adapt the model to their needs, for example by capturing more details or making it more abstract. For instance, to reason about waypointing (*i.e.* does a route traverse certain nodes) we may augment the routes with a set of traversed nodes (fig. 3).

```
let transFail transBase e (x : dict[edge, α]) =
  mapIte (fun e' -> e = e')
         (fun v -> None)
         (transBase e) x

let mergeFail mergeBase u x y =
  combine (mergeBase u) x y

let initFail initBase u = createDict (initBase u)
```

Figure 5. Meta-Protocol for Fault Tolerance Analysis

2.7 Programming New Analyses

An NV program does not necessarily have to emulate a routing protocol. Fault tolerance analysis is the holy grail of network verification tools: simulators cannot provide exhaustive guarantees, and SMT-based approaches do not scale. We implement — in just a few lines of NV code — a fault tolerance analysis that leverages sharing in our flexible map data structure to simulate all possible failure scenarios at once. Our analysis is orders-of-magnitude faster than the naive simulator-based approach of independently trying out all failure scenarios, and faster than SMT-based approaches while also scaling to large networks (*i.e.* with tens of thousands of links) that are out-of-reach to SMT verifiers (section 6).

The NV program of fig. 5 defines a *meta-protocol* that models routes under different *failure scenarios*, using a map from edges to routes. Intuitively, each map entry defines a single link failure scenario. The transfer function over an edge `e` uses an if-then-else operation over the map (see, section 3.1) to apply the transfer function of a regular routing protocol (*e.g.* the BGP transfer function `transBgp` of fig. 2a) over all map entries except the one which corresponds to `e`; that entry is replaced with a dropped route. Here we assume that a dropped route is denoted as a `None` value, but one can easily generalize the meta-protocol of fig. 5 to use different "default" values. The merge function uses the `combine` primitive to perform a pointwise application of the merge function for the underlying protocol.

The analysis can be easily extended to multiple link failures or even combinations of link and node failures. For instance, to compute the routes when there is a failed node and a link failure, we augment the map keys to be tuples of a node and an edge and adjust the transfer function to drop the route if it's being propagated over a failed link or from/towards a failed node.

The key insight behind our analysis is that, usually, a fault in the network only affects the routers that are topologically "close" to it. This is because networks are often built with some redundancy to sustain hardware faults. For instance, in the example of fig. 4 link failures inside a pod (a group of 4 routers, 2 pink and 2 blue in this case) do not affect the routes of nodes outside of it. More generally, faults can be organized in classes, each class triggering a different behavior. Unfortunately, determining those classes statically (*i.e.* without solving the routing problem) is a difficult open problem [22]. The key to the performance of our analysis is that it

```
d  ::= symbolic x : ty | require e | let x : ty = e | type t = ty
v  ::= ℕuℕ | true | false | None | Some v | (v1, v2) | {ℓ1 : v1; ...; ℓn : vn}
e  ::= v | x | let x : ty = e1 in e2 | fun (x : ty) → e | e1 e2 | Some e
       | (e1, e2) | {ℓ1 : e1; ...; ℓn : en} | e.ℓ | if e1 then e2 else e3
       | (match e0 with | p1 → e1 ... | pn → en)
ty ::= int⟨ℕ+⟩ | bool | node | edge | option[ty] | α | ty1 → ty2
       | (ty1, ..., ty2) | {ℓ1 : ty1, ..., ℓn : tyn} | dict[ty1, ty2]
```

Figure 6. Core NV syntax

```
create  : β → dict[α, β]
get     : dict[α, β] → α → β
set     : dict[α, β] → α → β → dict[α, β]
map     : (β → γ) → dict[α, β] → dict[α, γ]
mapIte  : (α → bool) → (β → γ) → (β → γ) → dict[α, β] → dict[α, γ]
combine : (β → β → γ) → dict[α, β] → dict[α, β] → dict[α, γ]
```

Figure 7. NV supported map operations.

dynamically finds these classes. It does so by leveraging the MTBDD implementation of maps which, by construction, groups together map entries with the same value.

3 The NV Language

NV has many features of a typical functional language (fig. 6), including let-bindings, match statements, (non-recursive) functions, and data structures such as options, tuples, records, and maps. NV also has a set data type, which is implemented as a map to boolean values. The base types are booleans, integers, nodes, and edges. Furthermore, integers are parametrized by a number of bits; for example, we write `int8` for the type of 8-bit integers and `5u8` for the 8-bit representation of the value 5. Lack of an annotation means a 32-bit size. Specifying the number of bits allows for more accurate modeling of protocol semantics and enables time and space savings in MTBDD-based analyses (section 5.1). NV also supports let-polymorphism, though the messages exchanged between nodes must have a concrete type.

An NV program is a series of declarations (`d`) capturing the topology of a network, the type of the routes exchanged, and the `init`, `transfer`, and `merge` functions that describe the semantics of the protocol (fig. 8).

NV also includes declarations to support verification. The first is an `assert` function, which expresses a specification about the converged state of a node. Users may also declare variables that are bound to a *symbolic value* [47]. A symbolic value represents a class of values as opposed to a single *concrete* value. In NV, their exact interpretation depends on the analysis. SMT-based analyses treat symbolic values as representing any concrete value. Analyses based on normalization (*e.g.* a simulator) require concrete, non-symbolic values. In this case, symbolics are treated as *inputs* to the program; prior to execution, symbolic values are fixed to concrete ones, provided by the programmer or by random generation.

A user may also constrain the class of concrete values that a symbolic value represents by providing a boolean *requires clause*. Any assignment of a concrete value to a symbolic one must ensure that the requires expression evaluates to true.

```
nodes    : int
edges    : set[edge]
init     : node → α
transfer : edge → α → α
merge    : node → α → α → α
assert   : node → α → bool
```

Figure 8. NV required declarations and types.

3.1 Semantics of Maps

One of the more interesting aspects of NV is its treatment of maps. Giving semantics to maps is straightforward (*e.g.* as a function), but efficiently implementing them in an interpreter or as constraints is not as easy. We found that *total maps* admit efficient implementations both when interpreted and when encoded as constraints.

The key principles driving our design of maps in NV are derived from their use in the context of routing:

1. Maps are typically indexed using statically-known values. For instance, routers route or block particular, known, subnets. They attach a particular tag to a BGP message. These values appear as constants in the configuration and need not be computed dynamically.
2. The final solutions of routing algorithms are maps from IP subnets to routes. Computing those solutions does not require aggregation operations, such as folds, across those maps. On the other hand, routing algorithms do need to compute routes for *all* subnets, so we do need operations that apply functions across all elements of the map.
3. There is a lot of symmetry in networks. Hence, many different keys may be associated with the same value. For instance, in our fault tolerance analysis the map keys represent many different failure scenarios, but the values are taken from an often small set of routes (*e.g.* as in fig. 4).

Points 1 and 2 are critical to our tuple-based SMT encoding of maps (section 5.2), while point 3 motivated our MTBDD-based interpretation of maps (section 5.1). In fact, we *enforce* point 1 by requiring that the keys used in map get/set operations are constants, rather than arbitrary expressions.

Figure 7 shows the map operations in NV. Map get (also written `m[k]`) and set (written `m[k := v]`) return and update the value associated with key `k` of the map, respectively. The `combine` function merges the values of two maps key-wise.

In line with the principles above, our maps support operations such as `map`, which maps a function over the values of a map, and `mapIte`, which maps one of two different functions over the values, based on a boolean function over their corresponding key. Intuitively, `mapIte` is useful to implement operations that would otherwise require the `map` operation to access both keys and values (*e.g.*, filtering routes based

on their prefix). While more general, this operation — commonly known as `mapi` — cannot be efficiently implemented using our MTBDD encoding of maps (section 5.1).

Tradeoffs and Limitations. When designing an intermediate verification language there is a tension between the expressivity needed to model the source language(s) and the tractability of analysis. In designing NV, we limited ourselves to constructs that admit efficient and complete verification procedures (section 5). In addition to our restrictions on map operations, we have omitted features such as recursive functions and datatypes. These conditions, while overly restrictive for a general-purpose language, can (with few exceptions) be easily accommodated in our context.

One such exception is BGP's record of a route's path as a string. At each hop, BGP will push the name of the router onto the front of the string. A user may construct route filters using regular expressions over the string. Such operations cannot be modeled in NV, though that may change in the future. Fortunately, however, *verification* (as opposed to implementation) does not usually require a precise model of such features—approximations suffice. For instance, one recent analysis [8] demonstrated such features can be over-approximated without loss of precision in real networks.

4 Translating Router Configurations

To obtain NV programs from actual router configurations, we modify Batfish to emit NV code. We first infer the basic structure of the network (the topology and which protocols are in use), then translate the configuration *route-maps* that implement network-specific policy.

4.1 Modeling the Topology and Active Protocols

In this stage, we infer the physical connectivity between routers, the different protocols that each router runs, and the prefixes (subnets) they announce.

For the topology, the translation is simple. We create one node in the graph for each router, and add edges between each pair for which Batfish has inferred physical connectivity. Representing the protocols is also straightforward. Each router in a network may run several routing protocols, and may also contain hardcoded *static* or *connected* routes. This information is stored in a table, known as a Routing Information Base (RIB), that holds information for all active protocols. The RIB also contains a *selection* of which protocol's route is best according to network policy.

We model the RIB as a map from prefixes (destinations) to a record containing each protocol's route to that destination. The example in fig. 9 maintains one each for OSFP and BGP, and two more for hardcoded routes. The fields of `ribEntry` are options because there may be no route through a given protocol. The `selected` field denotes the protocol whose route was chosen as best: 0 for OSFP, 1 for BGP, *etc.*

```
type ribEntry = {
  ospf      : option[ospfRoute];
  bgp       : option[bgpRoute];
  static    : option[staticRoute];
  connected : option[connectedRoute];
  selected  : option[int2] }

type ipv4Prefix = (int, int5)
type attribute = dict[ipv4Prefix, ribEntry]
```

Figure 9. Type of routes exchanged in our network model.

4.2 Modeling the Policy

Network operators implement policy over a network using a mechanism called *route-maps*. A route-map is a list of statements which test and modify certain characteristics of a route. For instance, a route-map applied on a BGP connection may check if a certain tag is present and increase or decrease the local preference value accordingly.

Abstractly, route-maps contain two types of statements: *conditional* statements which test properties of the route, and *mutation* statements which modify attributes of the route.

Intermediate Policy Representation. Converting route-maps to NV is complicated somewhat by the different abstraction levels – route-maps operate over a single route, while our NV encoding processes all routes at once using the `dict` data type. To convert route-maps to NV expressions, we go through a directed acyclic graph (DAG) based intermediate representation. We represent each route-map as a DAG in which non-leaf nodes correspond to conditional statements, and leaf nodes correspond to a list of mutation statements. This representation allows us to separate prefix processing (map keys) from route processing (map values) by swapping DAG nodes to reorder conditional statements.

Figure 10a shows a route-map that sets the local-preference value of a BGP route based on the attached communities and the destination prefix. The DAG of fig. 10b captures the semantics of this route-map; note that if no route-map matches, the route is implicitly dropped (denoted by $\perp$).

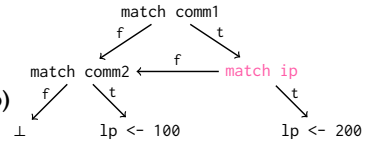
A natural translation from this DAG representation to NV is a chain of if-then-else expressions, which are mapped over routes in the RIB. However, since route-maps may test not only the route but also the prefix (*i.e.* the keys in the RIB), we must use the `mapIte` operation, which may test map keys. Doing so, however, requires some extra effort: conditional statements on the prefix must be executed first, so they may appear as predicates for `mapIte`.

Fortunately, our DAG-based IR makes this easy; we need only swap nodes in the DAG until all nodes which condition on the prefix are at the top (*i.e.* any parents also condition on the prefix). Figure 10c shows the result of applying this transformation to the DAG in fig. 10b. Translating to an NV expression using `mapIte` is easy now; the pink nodes are used as predicates to `mapIte`, while the rest are translated as if-then-else chains over the map values (fig. 10d).

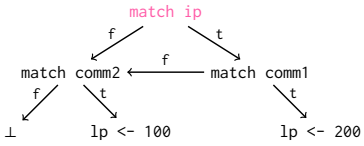
(a)

```
route-map RM1 permit 10
  match community comm1
  match ip address prefix-list pfx
  set local-preference 200
route-map RM1 permit 20
  match community comm2
  set local-preference 100
```

(b)



(c)



(d)

```
let map_option f o = match o with | None -> None | Some x -> f x

let transRouteMap (x : dict[ipv4Prefix,option[bgp]]) = mapIte (fun pre -> matchPrefix pre pfx)
        (fun ov -> map_option (fun v -> if matchComm comm1 v then Some {v with lp = 200} else
                                        if matchComm comm2 v then Some {v with lp = 100} else None) ov)
        (fun ov -> map_option (fun v -> if matchComm comm2 v then Some {v with lp = 100} else None) ov) x
```

Figure 10. (a) defines a single route-map, RM1; (b) shows its DAG representation. (c) shows the transformed DAG, where all conditional statements on prefixes are on the top of the DAG, and finally, (d) is what the resulting NV function looks like.

5 Network Analyses

5.1 Simulation

A control plane simulator is an algorithm that mimics the exchange and processing of routes as dictated by the semantics of the protocols in play. Existing simulators such as Batfish [17] and FastPlane [32] are invaluable industrial tools for testing the consequences of various routing configurations. The key difference between our simulator and these others is that they are designed to simulate specific protocols (*e.g.*, BGP, OSPF, etc) where as our simulator simulates the NV *programming language*. A standard interpreter implementation that treats NV structures as ordinary functional data structures would have led to non-competitive performance. To accelerate performance, we use specialized data structures (*e.g.*, MTBDDs) and domain-specific optimizations.

The Simulation Algorithm. Algorithm 1 presents the core simulation algorithm. We use the notation $[\![e]\!]$ to denote execution of the functional components of NV, including the init, transfer and merge functions. We describe the non-standard elements of the interpreter in the following subsection.

The goal of the algorithm is to compute the solution $\mathcal{L}$ to the system. Recall that such a solution is a *stable state* of the system. In other words, the solution $\mathcal{L}(v)$ at every node v must be equal to the merge of its initial attribute with all attributes transferred from its neighbors (see section 2.5).

Overall, the algorithm for computing solutions is a worklist algorithm that stores the nodes to be processed on a priority queue (q). Lines 6-9 initialize the solution $\mathcal{L}$ and populate the queue with all nodes in the network. Lines 10-11 select a node from the queue to process, or, if there are none left, terminate the algorithm.[2]

Lines 12-20 explain how to process a node u. To do so, u sends its current attribute to all of its neighbors v. Each neighbor v may need to update its solution. The neighbor v checks whether it has previously received information from u (*i.e.*, if $u \in \text{received}(v)$). If not, it can simply merge the new information with its current solution (lines 19-20). Otherwise, v's solution contains stale information from u. The simple thing to do in this case is to recompute a merge of all received messages, including the new one from u (line 18). However, this is costly, so we use an observation from ShapeShifter [8], which is that when (`merge old new`) = `new` it suffices to merge `new` into the existing solution rather than recomputing a merge of all received messages (lines 15-17).

[2]The algorithm is not guaranteed to terminate. Past work [12, 24, 45] has studied criteria for termination that can be checked.

Interpreting NV with MTBDDs. Multi-Terminal binary decision diagrams (MTBDDs) are a variant of BDDs capable of representing functions from finite domains to arbitrary values (rather than just to booleans). MTBDDs have been used for symbolic model checking of domains requiring rich structure, such as probabilistic systems [2]. We leverage the fact that MTBDDs store one copy of each leaf — since NV maps typically contain many repeated values, an MTBDD representation is quite compact. Moreover, the NV simulator frequently applies the same operation to each entry in a map — sharing of leaves means we need apply the operation only once per distinct entry. Finally, the canonical structure of MTBDDs enables efficient equality tests. Fast (in)equality tests significantly improve simulator performance by quickly testing if a node's attribute changed after a merge operation.

To represent a map as an MTBDD, we must represent its domain as a series of binary decisions. With the exception of maps and functions, types in NV are designed to be finitary, hence they can be represented in such a fashion. For instance, finite integers are represented bitwise; fig. 11a shows the MTBDD corresponding to a total map from 3-bit integers to values of type `option[int]`.

Most map operations reduce to constructing BDDs and combining them with the MTBDD that represents the map. Figure 11 illustrates how to construct several MTBDDs for simple functions over 3-bit integers and apply them. Although MTBDDs provide us many advantages, constructing BDDs and combining them can be computationally expensive. To amortize the cost of these operations we cache them; in practice, cache hits are likely to occur frequently during simulation since multiple nodes have similar configurations (*e.g.* filtering the same communities).

Native Simulation. Past implementations of control plane simulators ([17, 32]) have *interpreted* computational elements

Algorithm 1 Network Simulator

```
1:  procedure UPDATE(L, q, v, route)
2:    if route ≠ L(v) then L(v) ← route; q ← q ∪ {v}
3:
4:  procedure SIMULATE(V, E, init, trans, merge)
5:    q ← {}
6:    for u ∈ V do
7:      L(u) ← init(u)                          ▷ Best route of node u
8:      received(u)(u) ← init(u)                ▷ Routes received at u
9:      q ← q ∪ {u}
10:   while q ≠ empty do
11:     u ← pop q                               ▷ Propagate u's route
12:     for v ∈ neighbors(u) do
13:       new ← ⟦trans((u, v), L(u))⟧
14:       if u ∈ received(v) then               ▷ Is there a stale route?
15:         old ← received(v)(u)
16:         if ⟦merge(v, old, new)⟧ = new then  ▷ Incremental update
17:           UPDATE(L, q, v, ⟦merge(v, L(v), new)⟧)
18:         else UPDATE(L, q, v, ⋃⟦merge⟧ received(v))   ▷ Full update
19:       else UPDATE(L, q, v, ⟦merge(v, L(v), new)⟧)
20:       received(v)(u) ← new
```

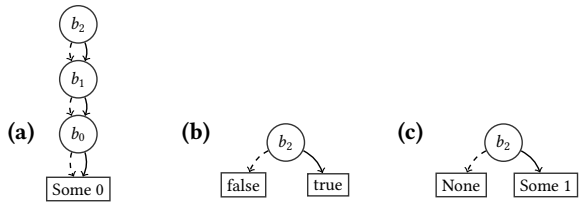


Figure 11. Implementation of `mapIte (fun k -> k > 3) opt_incr (fun v -> None) (create (Some 0))`, which increments the length of routes with key greater than 3, and drops others, for a map from 3-bit integers to the `Some 0` value. (a) MTBDD for a map (`create (Some 0)`) (presented is an unreduced MTBDD). Nodes are labelled with a bit (b_2 is most significant). If a bit is false, one follows the dashed line to find the corresponding structure; otherwise, the solid line. Here, any bit pattern leads to `(Some 0)`. (b) MTBDD encoding of the `(fun k -> k > 3)`. (c) The result of `mapIte`, by performing an MTBDD apply operation on (a) and (b) and mapping `opt_incr` and `(fun v -> None)` over the result.

of control plane protocols. Of course, interpreters are generally much slower than *compiled* programs. In our case, because we have already mapped the ad hoc vendor configuration languages into a functional programming language, the work required to compile NV is significantly reduced. Indeed, we can translate NV's computational core to OCaml, use OCaml's compiler to obtain assembly code, and link the compiled binary to the simulator.

The translation from NV to OCaml is straightforward for most constructs; functions, options, integers, booleans, and tuples all have corresponding constructs in OCaml. The notable exception is maps, as there is no MTBDD-based map construct in OCaml, and so we must reuse the MTBDD library we developed for the interpreter. This library converts the abstract syntax of NV values into keys for MTBDDs and allows NV values to be stored in the leaves of MTBDDs. However, it does not allow arbitrary OCaml values to be used as keys/values in MTBDDs. To bridge the gap between this library and the OCaml representation of the rest of the NV values, we convert a subset of OCaml into NV (an *embedding*) and, conversely, convert the subset of NV that may be used as map keys/values back into OCaml (an *unembedding*).

5.2 SMT-based verification

The key insight in SMT verification of control planes is that one does not need to model the message-passing process directly[6, 24]. Instead, one can specify the requirements on the stable state—that is, that every node holds an attribute equal to the merge of its initial state and the transfer of its neighbor's attributes. If the converged states are specified by a formula `N` (shown in section 2.5) and we wish to verify a property `P` holds in those states, it suffices to show that `N` $\wedge \neg$`P` is unsatisfiable; that is, there is no converged state of our network in which `P` does not hold.

As with the simulator, the key difference between NV's SMT verification engine and previous network verification engines, like the one implemented in MineSweeper, is the flexibility of the system. Rather than encoding specific protocols directly as SMT formulae, we encode features of the NV language. Most importantly, we can optimize NV programs far more systematically than was possible during the ad hoc, one-pass translation used in *e.g.* MineSweeper.

An Optimizing Pipeline to SMT. SMT solvers offer theories for reasoning about complex constructs such as datatypes and arrays. Unfortunately, we have found that such theories perform relatively poorly in this context. As a result, before converting NV to SMT, we eliminate complex NV data types through a series of mostly standard source-to-source transformations.

Map Unrolling. Map operations are chosen to allow programmers to specify maps with huge domains, such as the domain of all 2^{32} IP addresses, but to only pay a cost proportional to the subset that is used, such as the (much smaller) set of IP addresses that appear in the network. In particular, because there are no aggregation operations over the the entire map, such as a fold, entries that are never accessed via a `get` operation need not be represented. We implement such sparse maps as tuples, via a *map unrolling* process, where an element of the tuple is reserved for each key that accesses the map. Map accesses then become tuple projections.

When the keys used to access a map are constants, implementing the map as a tuple is straightforward — collect all the n constant keys used in the program and create a n-tuple with an entry for each constant. However, we also allow indexing maps with *symbolic values*, whose value is

unknown at compile time. To accommodate both n constant keys $c_1, \ldots, c_n$ and m symbolic keys $s_1, \ldots, s_m$, we use a tuple of size $n + m$ where the value associated with c_i appears in element i and the value associated with s_j appears in element $n + j$. Of course, symbolic key s_j may actually resolve to the constant c_i. In this case, the computation must be as if the constant c_i was used in s_j's place. To do this, we encode map get on a symbolic key as follows (map set is similar). Here, assume we have one constant key `c` and two symbolic keys `s1, s2`.

```
encode(m[s]) = if s = c then encode(m).0 else
  if s = s1 then encode(m).1 else encode(m).2
```

Option Unboxing. Expressions of type `option[A]` are translated to pairs of type `(bool, A)` where the first component is `false` for `None` (and the second component is irrelevant), and `true` for `Some`.

Tuple Expansion and Flattening. After maps and options have been eliminated, the only complex data type left are tuples. We eliminate them too by flattening nested tuples and expanding variables of tuple type, and then simply encode the flat tuples as independent variables/expressions.

Partial Evaluation. Our transformations often lead to an explosion in program size as they introduce many intermediate expressions. To mitigate this, we partially evaluate the program and apply some additional simplifications before SMT encoding, normalizing away most of the clutter introduced by language abstractions and transformations.

From Expressions to Constraints. To translate an NV program to constraints, we follow a standard model checking approach: we inline all functions and rename variables to ensure that bindings are unique. After applying the transformations above, the remaining expressions have a direct translation to SMTLIB2. Importantly, the translation only relies on the quantifier-free core and linear arithmetic (or bitvector) fragments of SMT solvers, which helps achieve good solver performance and guarantees completeness.

6 Evaluation

To evaluate NV[3], we conducted experiments on several benchmarks, including a collection of real data center and wide-area network topologies along with a collection of synthetic router configurations based on real policies. We evaluate NV along several dimensions: (i) the performance of its symbolic SMT-based analysis compared to MineSweeper [6], (ii) the performance of the new fault-tolerance analysis compared to MineSweeper, (iii) the performance of its simulation engine compared to Batfish [17], and (iv) the impact of native compilation on network analysis time. We run all experiments on a 2015 Mac with a 4Ghz i7 CPU and 16GB of memory.

[3]The source code for NV and the networks used for experiments can be found at https://github.com/NetworkVerification/nv

6.1 Networks Studied

For data center topologies, we focus on FatTree [1] designs, commonly used to interconnect large numbers of servers while providing fault tolerance and high bisection bandwidth. Fattree designs are parameterized by k, the number of "pods", and by varying k, one can explore the impact of topology size on analysis time. For routing, modern datacenter designs use the eBGP routing protocol [29] for its scalability and policy-rich configurability, coupled with variants of shortest path routing and equal cost multipath (ECMP) load balancing.

We consider two different policies described in the literature: a pure shortest-path routing policy (denoted using `SP`), and a variant that uses tagging and filtering to disallow "valley routing"' [9], *i.e.* dropping routes that go through the same layer of the fat tree multiple times (denoted `FAT`).

To evaluate our fault tolerance, we consider both the (symmetric) data center networks as well as an asymmetric wide-area network topology (USCarrier, consisting of 174 nodes and 410 links) from the Topology Zoo [28] with a policy previously synthesized by NetComplete [13].

6.2 Performance of SMT Verification

We compare our SMT-based analysis with MineSweeper, the state-of-the-art SMT verifier for control planes. For the experiment we use six FatTree networks running the routing policies described above, and ranging in size from 80 to 180 routers (nodes)[4]. MineSweeper and NV solve slightly different problems; Minesweeper asserts facts about a data packet in the network, which requires modeling the part of the control plane that affects this data packet. On the other hand, NV currently only models the control plane and does not consider data packets. However, since there are no dataplane access control lists (ACLs) in the examples, the encodings are similar [6]. Each leaf in the fat tree announces a destination prefix; we pick a node at random (called the *destination*). For MineSweeper, we assert that a data packet sent from

[4]SP(k) and FAT(k) each have $(5/4)k^2$ nodes and k^3 edges

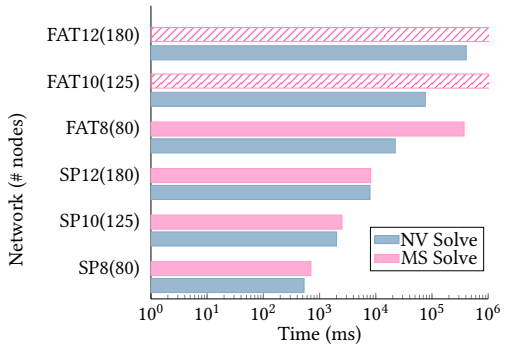


Figure 12. SMT time to solve the constraints for NV and MineSweeper (MS). MineSweeper timeouts after 30 minutes for FAT10 and FAT12.

any node in the network, with a concrete destination IP that matches the prefix of our destination, reaches the destination. Similarly, in NV, we assert that every node has a route to the prefix announced by the destination node.

Discussion. Figure 12 compares the SMT time for the constraints generated by MineSweeper and NV. Our goal is not to compare the absolute verification time, but rather to observe differences in performance trends.

Networks based on shortest-path routing have similar verification time and scaling pattern. The FAT networks, which route based on more complex policy, provide more interesting data. For NV, SMT time is 40-50x slower compared to the SP networks. On the other hand, MineSweeper only verified the smaller 80 node network, with a slowdown of more than 500x. It is difficult to pinpoint the cause of this difference between the two; however, it is likely that some of the optimizations MineSweeper performs do not kick in when dealing with more complex policy. For instance, MineSweeper also performs some forms of partial evaluation. However, unlike NV, MineSweeper reduction rules are rather ad-hoc, as they are defined over a language that was designed for neither partial-evaluation nor translation to constraints.

Unsurprisingly, MineSweeper computes the SMT encoding faster than NV (not shown in fig. 12). This is because MineSweeper builds on top of the original structure of the problem, while NV requires many transformations to reduce the abstractions introduced. However, the bottleneck remains the constraint solver; the encoding procedure can be further optimized, and even parallelized, but it is not obvious how to improve the performance of the SMT solver.

6.3 Performance of Fault Tolerance Analysis

Next, we examine the performance of our map-based fault tolerance analysis that simulates all faults at once.

Comparison with SMT-based Approaches. Currently, the only other tool that provides exhaustive fault-tolerance verification of networks with expressive policy is the SMT-based analysis of MineSweeper. Figure 13a compares the verification performance of our analysis; MineSweeper; and the SMT backend of NV, when verifying single link fault tolerance. As witnessed in the previous experiment, SMT-based techniques have certain scaling limits even when failures — which largely increase the state space — are not considered. Unsurprisingly, in the presence of failures the performance of the SMT-based analysis deteriorates even faster before eventually timing out. In contrast, our MTBDD-based analysis leverages the symmetries in failure scenarios and computes the routes for any possible failure in a matter of seconds.

Failure Analysis Scaling. We further evaluate how the fault tolerance analysis scales as we increase the size of the networks and the number of failures (fig. 13b). We note that precise fault tolerance analysis of some of these networks is out of reach of existing network analysis tools[5].

In the highly symmetric fat tree topologies, the analysis scales linearly with the number of link failure combinations. For instance, considering a single link failure, scaling is practically linear in the number of links. Of course, the number of unique failure combinations grows exponentially as we increase the bound on link failures, as demonstrated by the slowdown when considering 2 or 3 link failures for a network like FAT28, which has roughly 22,000 links.

On the other hand, USCarrier faces greater impact when the number of failures is increased; this is because the network is less symmetric and lacks redundancy to sustain multiple failures. As more edges fail, the network's behavior changes significantly. Hence, by the time we reach 3 link failures, the routes computed for each scenario can vary wildly, reducing the sharing that MTBDDs exploit.

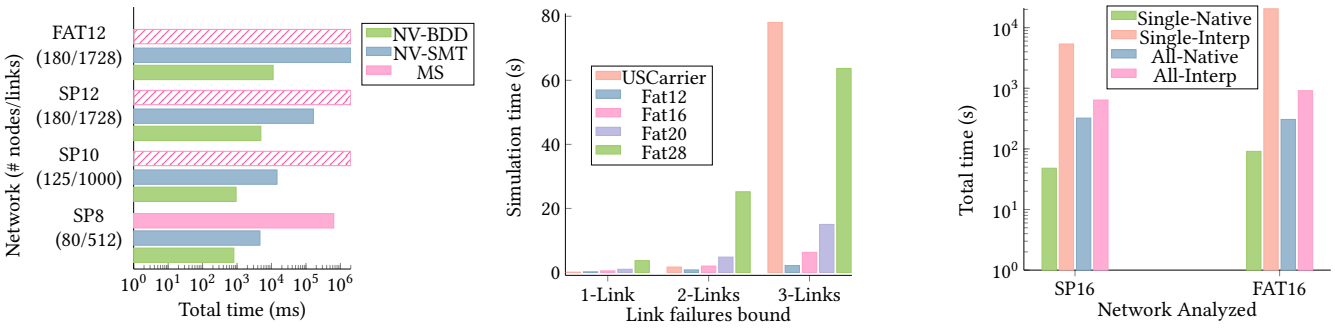
Single-prefix vs All-prefixes. An alternative model takes advantage of disjoint prefixes, *i.e.* prefixes whose routing solutions can be computed independently. We found that fault tolerance analysis for each destination prefix separately is more effecient than doing all-prefixes simultaneously, for a number of reasons. First, single-prefix models have significantly more uniform routes among different failure scenarios and hence the underlying MTBDDs achieve better sharing. Second, single-prefix models make the most of native execution thanks to a reduced number of embedding/unembedding operations (see next section for details). Third, we amortize the compilation cost of native simulation, as we compile once and then run multiple simulations, one per destination. In addition, we could run the simulations for different destinations in parallel, further reducing execution time.

Figure 13c shows the total execution time for these choices. In particular, it compares the performance of the fault tolerance analysis for 1 link failure over each prefix independently vs. all-prefixes simultaneously. The networks used are SP16 and FAT16. A total of 128 destinations are announced in these networks, hence the single prefix analysis was run 128 times (but compiled once). Doing the analysis on each prefix separately using the native simulator resulted in 3-7x times speedup (with no parallelism) compared to doing all prefixes simultaneously. We discuss the performance of the interpreter and native execution in the next section.

6.4 Simulation Performance

To gauge the performance of our simulator, we measure how it fares compared to Batfish when solving the all-prefixes routing problem for networks ranging in size from 500 to 1280 nodes. Batfish is written in Java, and employs a parallelized simulation engine. Figure 14 shows that our MTBDD-based simulator is an order of magnitude faster than Batfish. More importantly, as the network size grows, the runtime

[5] ARC [21] can scale, but is not compatible with rich policy (*e.g.*, tagging)



(a) **(b)** **(c)**

Figure 13. (a) shows the total time of our fault tolerance analysis and the SMT approaches of NV and MineSweeper for a single-prefix. (b) shows how our fault tolerance analysis scales (excludes compilation time) as the size of the network and the failures increase. (c) compares the total time (including compilation time) to do fault tolerance analysis over all prefixes simultaneously or each prefix separately.

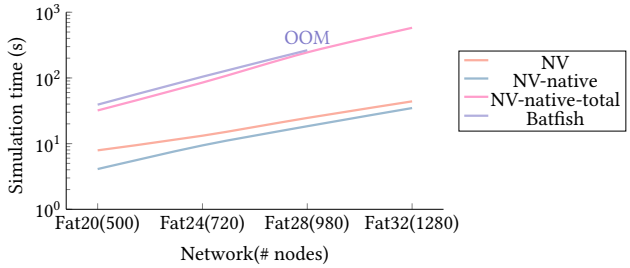


Figure 14. All-prefixes analysis time (Batfish runs out-of-memory after FAT28). NV uses the interpreter for simulation. NV-native uses the native simulator but excludes compilation time. NV-native-total includes OCaml's compilation time.

of NV only marginally increases, while Batfish has a steeper trend. The same is true of the memory consumption of the two; NV peaks at 2GB for `FAT32` while Batfish runs out of memory (16GB) on the smaller `FAT28`. Although Batfish's route model is more detailed, these performance differences are mainly attributed to the MTBDD representation of a router's RIB in NV; these networks advertise hundreds of prefixes and the ability to compactly represent them and process them in bulk is key to scaling the simulation.

Native and Interpreted Simulation. Native execution can deliver significant performance improvements compared to conventional interpreter-based simulators, but it also has inherent overheads. Compiling a large OCaml program to assembly is a time consuming process, sometimes more so than the simulation itself. Furthermore, embedding and unembedding values between NV and OCaml also induces some overhead, although we try to minimize that through caching.

In the all-prefix simulations of fig. 14, execution time of the native simulator is constrained by embedding/unembedding operations on maps. As the policy is rather simple, the overhead of these operations dominates execution time. On the other hand, if more complex operations are used, native simulation can significantly outperform the interpreted one. An example of this is our fault tolerance analysis (fig. 13c, orange and green bars): the functions applied over maps are more complex (applying a full transfer function) and thus the embedding operations are amortized. Overall, it makes sense to use the native simulator in applications where the benefits of faster execution outweigh the overheads of compilation, such as simulating networks with complex policy, running analyses like fault tolerance, and/or doing multiple simulations by instantiating symbolic values with different concrete values which amortizes the compilation cost.

7 Related Work

Analysis of Data Planes. Over the past decade, researchers have developed many systems for analysis or verification of network data planes. Examples include systems such as Anteater [33], Header-Space Analysis (HSA) [26], NoD [31], and Veriflow [27]. These systems operate by pulling snapshots of the current data plane from a set of routers and then checking that the data plane exhibits key properties such as reachability, isolation, or absence of black holes.

NetKAT [4] is another line of work in this vein. NetKAT offers a rich specification language based on Kleene Algebra with Tests (KAT). The operators supplied by NetKAT suffice to encode a network's topology as well as the forwarding behavior of its data plane. However, NetKAT cannot encode the semantics of network control plane protocols such as BGP. Such protocols compare routes to one another, discarding some and forward others, and they continue executing until they find a set of "stable paths."

McNetKAT is an extension of NetKAT with probabilistic operations [18, 44]. Such probabilistic operations make it possible to reason about expected congestion and probabilistic failures. Indeed, McNetKAT has studied verification of *data plane* fault tolerance properties extensively. Their system also leverages variants of BDDs to scale their analysis to the point of being able to analyze data centers with a few hundred nodes in a few minutes [44]. The work in this paper on *control plane* fault tolerance properties complements

the work on McNetKAT rather than competing with it. In addition to differing by virtue of analyzing the control plane rather than the data plane, our work adopts a different failure model (we consider a bounded number of failures rather than an arbitrary number of probabilistic failures).

It is also possible to extend data plane models with stateful primitives [3, 5, 36, 39, 48]. These stateful models can represent various kinds of middle boxes including NATs or stateful firewalls. Bayonet [19] combines probabilistic primitives with stateful operations. Gehr *et al.* [19] show that these stateful systems can be also used to model control plane protocols such as OSPF. However, analysis of these stateful systems appear to scale much more poorly than NV or other tools designed specifically for control plane analysis. For instance, it takes Bayonet several minutes to analyze networks that have just 30 nodes [19].

Modeling Control Planes. Batfish [17] is a network analysis framework that parses network configuration files, simulates control plane execution, generates a dataplane and then runs a dataplane analysis over that data plane. One of the benefits of using Batfish over simpler data-plane-only analysis tool kits of the kind mentioned in the prior paragraphs, is that Batfish allows a user to ask "what if" questions: What if this link were to fail? By simulating control plane semantics afresh under such speculative conditions, Batfish can answer those questions. The data plane tools that simply download the current dataplane from a set of routers cannot. Batfish also serves as a platform for developing new control plane analyses. Tools such as ARC [21], MineSweeper [6], ShapeShifter [8], and Bonsai [7] have all been implemented on top of Batfish. One of the central contributions of NV is to make the process of constructing these auxiliary tools simpler by creating a more compact, compositional and expressive intermediate language for representing network protocols. Of course, NV continues to depend upon Batfish for its front end. Moreover, unlike Batfish, NV does not compute a data plane, nor does it support protocols like iBGP that rely on data plane elements.

ARC [21] is another system that may be viewed as a modeling language for network control planes. ARC's models are graphs and it performs network analysis by executing standard graph algorithms, such as shortest paths, over these graphs. One of the advantages of this approach is that many graph algorithms are guaranteed to be polynomial, whereas the BDDs and SAT/SMT encodings used by NV result in exponential algorithms in the worst case. On the other hand, ARC is incapable of representing certain control plane features, such as BGP local preference and communities.

Routing algebras [45] and metarouting [25] are another line of work on building models of network control planes. The key difference between these systems and NV lies in the properties considered and the approach to checking them. The work on Metarouting was concerned primarily with convergence of protocols. Moreover, instead of developing tools to check whether existing networks satisfy the properties of interest, the work provides local, topologically-independent criteria of the transfer function that imply various convergence properties of the protocol. Additionally, it devises combinators designed to ensure those key properties are preserved when constructing new protocols. In contrast, NV focuses on analyzing global, end-to-end properties of networks that depend upon a combination of network topology and policy such as reachability and fault-tolerance. We use SMT and BDD-based verification to validate these properties and we have built tools that translate real configurations into our declarative modelling language.

Other Control Plane Fault Tolerance Analyses. As mentioned above, ARC [21] may be viewed as a modelling language for control plane protocols. Like NV, one of the focuses of ARC is fault tolerance analysis, which they reduce to standard graph algorithms. Origami [22] is another fault tolerance analysis. It operates by exploiting symmetries in the network's policy and topology to reduce its size. It computes an abstraction of the network using a counterexample-guided abstraction refinement algorithm that uses an oracle to determine whether the property holds. An SMT verifier was used as an oracle, but there is no reason that an MTBDD-based analysis could not be used instead. Like ARC, Origami also imposes restrictions on policy (though the restrictions are less stringent). Furthermore, its abstraction only preserves an approximation of the original routes, making it unsuitable for reasoning about properties beyond reachability.

Verification Languages. NV was inspired, in part, by the Rosette solver-aided language [47]. Rosette embeds a symbolic compiler in the Racket programming language to translate solver-aided programs into logical constraints that can be solved by SMT solvers. Like Rosette, NV is a functional language with symbolic values and constraints. However, NV differs from Rosette in both the tools it supports (*e.g.*, MTBDDs) and the applications (networking) it attacks.

NV also draws inspiration from intermediate verification languages such as Boogie [30], Why3 [16], and CIVL [43]. However, while these tools focus on general purpose programming languages and the complications that come with them (*e.g.*, recursion), NV is specialized for the data structures, functions, and algorithms needed to analyze networks.

Acknowledgments

We would like to thank the anonymous reviewers and our shepherd Petar Tsankov whose feedback greatly improved this paper. This work was supported in part by the National Science Foundation awards NeTS 1703493 and FMitF 1837030, and a Facebook Research Award on "Network control plane verification at scale".

References

[1] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. 2008. A Scalable, Commodity Data Center Network Architecture. In *SIGCOMM*.

[2] Luca de Alfaro, Marta Z. Kwiatkowska, Gethin Norman, David Parker, and Roberto Segala. 2000. Symbolic Model Checking of Probabilistic Processes Using MTBDDs and the Kronecker Representation. In *Proceedings of the 6th International Conference on Tools and Algorithms for Construction and Analysis of Systems: Held As Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS 2000 (TACAS '00)*. 395–410.

[3] Kalev Alpernas, Roman Manevich, Aurojit Panda, Mooly Sagiv, Scott Shenker, Sharon Shoham, and Yaron Velner. 2018. Abstract Interpretation of Stateful Networks. In *International Symposium on Static Analysis (Lecture Notes in Computer Science)*, Andreas Podelski (Ed.), Vol. 11002. Springer, 86–106.

[4] Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. 2014. NetKAT: Semantic Foundations for Networks. In *POPL*.

[5] Mina Arashloo, Yaron Koral, Michael Greenberg, Jennifer Rexford, and David Walker. 2016. SNAP: Stateful Network-Wide Abstractions for Packet Processing. In *ACM SIGCOMM*.

[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 *(SIGCOMM '18)*. 476–489.

[8] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. 2020. Abstract interpretation of distributed network control planes. In *Proceedings of the ACM on Programming Languages*, Vol. 4. Article 42.

[9] Ryan Beckett, Ratul Mahajan, Todd Millstein, Jitendra Padhye, and David Walker. 2016. Don'T Mind the Gap: Bridging Network-wide Objectives and Device-level Configurations. In *SIGCOMM*.

[10] Cisco. 2019. Cisco IOS Master Command List, All Releases. https://www.cisco.com/c/en/us/td/docs/ios-xml/ios/mcl/allreleasemcl/all-book.html.

[11] Edmund M Clarke, Masahiro Fujita, and Xudong Zhao. 1996. Multi-terminal binary decision diagrams and hybrid decision diagrams. In *Representations of discrete functions*. Springer, 93–108.

[12] Matthew L. Daggitt, Alexander J. T. Gurney, and Timothy G. Griffin. 2018. Asynchronous Convergence of Policy-rich Distributed Bellman-ford Routing Protocols. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*. 103–116.

[13] Ahmed El-Hassany, Petar Tsankov, Laurent Vanbever, and Martin Vechev. 2018. NetComplete: Practical Network-Wide Configuration Synthesis with Autocompletion. In *USENIX NSDI'18*. Renton, WA, USA.

[14] 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*.

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

[16] Jean-Christophe Filliâtre. 2003. Why: a multi-language multi-prover verification tool. (2003).

[17] 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*.

[18] Nate Foster, Dexter Kozen, Konstantinos Mamouras, Mark Reitblatt, and Alexandra Silva. 2016. Probabilistic netkat. In *European Symposium on Programming*. Springer, 282–309.

[19] Timon Gehr, Sasa Misailovic, Petar Tsankov, Laurent Vanbever, Pascal Wiesmann, and Martin Vechev. 2018. Bayonet: probabilistic inference for networks. *ACM SIGPLAN Notices* 53, 4 (2018), 586–602.

[20] Aaron Gember-Jacobson, Aditya Akella, Ratul Mahajan, and Hongqiang Harry Liu. 2017. Automatically repairing network control planes using an abstract representation. In *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, 359–373.

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

[22] Nick Giannarakis, Ryan Beckett, Ratul Mahajan, and David Walker. 2019. Efficient verification of network fault tolerance via counterexample-guided refinement. In *International Conference on Computer Aided Verification*. Springer, 305–323.

[23] Joanne Godfrey. 2016. The Summer of Network Misconfigurations. https://blog.algosec.com/2016/08/business-outages-caused-misconfigurations-headline-news-summer.html.

[24] Timothy G. Griffin, F. Bruce Shepherd, and Gordon Wilfong. 2002. The Stable Paths Problem and Interdomain Routing. *IEEE/ACM Trans. Networking* 10, 2 (2002).

[25] Timothy G. Griffin and João Luís Sobrinho. 2005. Metarouting. In *Proceedings of the 2005 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM '05)*. 1–12.

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

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

[28] Simon Knight, Hung X Nguyen, Nickolas Falkner, Rhys Bowden, and Matthew Roughan. 2011. The internet topology zoo. *IEEE Journal on Selected Areas in Communications* 29, 9 (2011), 1765–1775.

[29] P. Lapukhov, A. Premji, and J. Mitchell. 2015. Use of BGP for routing in large-scale data centers. Internet draft.

[30] K Rustan M Leino. 2008. This is boogie 2. *manuscript KRML* 178, 131 (2008), 9.

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

[32] Nuno P Lopes and Andrey Rybalchenko. 2019. Fast BGP Simulation of Large Datacenters. In *International Conference on Verification, Model Checking, and Abstract Interpretation*. Springer, 386–408.

[33] Haohui Mai, Ahmed Khurshid, Rachit Agarwal, Matthew Caesar, P. Brighten Godfrey, and Samuel Talmadge King. 2011. Debugging the Data Plane with Anteater. In *SIGCOMM*.

[34] Hugo Martin and Samantha Masunaga. 2015. United Airlines blames grounding of hundreds of flights on computer glitch. https://www.latimes.com/business/la-fi-united-flights-grounded-20150708-story.html.

[35] Kieren McCarthy. 2019. BGP super-blunder: How Verizon today sparked a 'cascading catastrophic failure' that knackered Cloudflare, Amazon, etc. https://www.theregister.co.uk/2019/06/24/verizon_bgp_misconfiguration_cloudflare/.

[36] Jedidiah McClurg, Hossein Hojjat, Nate Foster, and Pavol Černỳ. 2016. Event-driven network programming. *ACM SIGPLAN Notices* 51, 6 (2016), 369–385.

[37] Ben Mutzabaugh. 2016. Unions want Southwest CEO removed after IT outage. https://www.usatoday.com/story/travel/flights/todayinthesky/2016/08/01/unions-want-southwest-ceo-removed-after-outage/87926582/.

[38] Option to select BGP path ranking criteria [n.d.]. Option to select BGP path ranking criteria. https://github.com/batfish/batfish/pull/2076.

[39] Aurojit Panda, Ori Lahav, Katerina J. Argyraki, Mooly Sagiv, and Scott Shenker. 2017. Verifying Reachability in Networks with Mutable Datapaths. In *USENIX Symposium on Networked Systems Design and Implementation*, Aditya Akella and Jon Howell (Eds.). USENIX Association, 699–718.

[40] B. Quoitin and S. Uhlig. 2005. Modeling the Routing of an Autonomous System with C-BGP. *Netwrk. Mag. of Global Internetwkg.* 19, 6 (November 2005), 12–19.

[41] Simon Sharwood. 2016. Google cloud wobbles as workers patch wrong routers. http://www.theregister.co.uk/2016/03/01/google_cloud_wobbles_as_workers_patch_wrong_routers/.

[42] Jonathan Shieber. 2019. Facebook blames a server configuration change for yesterday's outage. https://techcrunch.com/2019/03/14/facebook-blames-a-misconfigured-server-for-yesterdays-outage/.

[43] Stephen F Siegel, Manchun Zheng, Ziqing Luo, Timothy K Zirkel, Andre V Marianiello, John G Edenhofner, Matthew B Dwyer, and Michael S Rogers. 2015. CIVL: the concurrency intermediate verification language. In *SC'15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–12.

[44] Steffen Smolka, Praveen Kumar, David M Kahn, Nate Foster, Justin Hsu, Dexter Kozen, and Alexandra Silva. 2019. Scalable verification of probabilistic networks. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 190–203.

[45] João Luís Sobrinho. 2005. An Algebraic Theory of Dynamic Network Routing. *IEEE/ACM Trans. Netw.* 13, 5 (October 2005), 1160–1173.

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

[47] Emina Torlak and Rastislav Bodik. 2013. Growing solver-aided languages with rosette. In *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software*. ACM, 135–152.

[48] Yaron Velner, Kalev Alpernas, Aurojit Panda, Alexander Rabinovich, Mooly Sagiv, Scott Shenker, and Sharon Shoham. 2016. Some Complexity Results for Stateful Network Verification. In *Tools and Algorithms for the Construction and Analysis of Systems (Lecture Notes in Computer Science)*, Marsha Chechik and Jean-François Raskin (Eds.), Vol. 9636. Springer, 811–830.

[49] Konstantin Weitz, Doug Woos, Emina Torlak, Michael D. Ernst, Arvind Krishnamurthy, and Zachary Tatlock. 2016. Formal Semantics and Automated Verification for the Border Gateway Protocol. In *NetPL*.