Modular Control Plane Verification via Temporal Invariants

ANONYMOUS AUTHOR(S)

Monolithic control plane verification cannot scale to hyperscale network architectures with tens of thousands of nodes, heterogeneous network policies and thousands of network changes a day. Instead, *modular verification* offers improved scalability, reasoning over diverse behaviors, and robustness following policy updates. We introduce Timepiece, a new modular control plane verification system. While one class of verifiers, starting with Minesweeper [Beckett et al. 2017a], were based on analysis of stable paths, we show that such models, when deployed naïvely for modular verification, are unsound. To rectify the situation, we adopt a routing model based around a logical notion of time and develop a sound, expressive, and scalable verification engine.

Our system requires that a user specifies interfaces between module components. We develop methods for defining these interfaces using predicates inspired by temporal logic, and show how to use those interfaces to verify a range of network-wide properties such as reachability or access control. Verifying a prefix-filtering policy using a non-modular verification engine times out on an 80-node fattree network after 2 hours. However, Timepiece verifies a 2,000-node fattree in 2.37 minutes on a 96-core virtual machine. Modular verification of individual routers is embarrassingly parallel and completes in seconds, which allows verification to scale beyond non-modular engines, while still allowing the full power of SMT-based symbolic reasoning.

1 INTRODUCTION

Today, in virtually every facet of our daily lives, we interact with networking infrastructure. Unreliable or failed infrastructure may lock us out of an ATM, a virtual court hearing, or even calling emergency services [Evans 2022]. Network operators program this infrastructure using distributed routing protocols, where each router in a network may run thousands of lines of configuration code. Routine configuration updates may inadvertently render a router unreachable [Strickx and Hartman 2022] or violate isolation requirements that prevent flooding [Vigliarolo 2022]. With networking now ubiquitous, major cloud providers are seeing sustained financial growth in response to mounting demand for reliable networking [Miller 2022]. These cloud networks may have hundreds of data centers, each with hundreds of thousands of devices running thousands of heterogeneous policies, and receiving thousands of updates every day [Jayaraman et al. 2019b]. This demand suggests a commensurate *infrastructure growth* will also take place as networks accommodate more and more users: with this come greater perils when configuration updates go wrong.

To safeguard against these dangers, operators can use *control plane verification* techniques to analyze their networks [Abhashkumar et al. 2020; Beckett et al. 2017a, 2018, 2019; Fayaz et al. 2016; Gember-Jacobson et al. 2016; Lopes and Rybalchenko 2019; Prabhu et al. 2017; Weitz et al. 2016; Ye et al. 2020]. Until recently, however, research has focused on *monolithic* verification of the entire network at once, which is not feasible for large cloud provider networks. Such networks demand *modular* techniques that can divide the network into components to verify in isolation. This approach has proven successful in the context of software verification [Alur and Henzinger 1999; Flanagan and Qadeer 2003; Giannakopoulou et al. 2018; Grumberg and Long 1994; Henzinger et al. 1998] and network data plane verification [Jayaraman et al. 2019a]. The *interfaces* between network components must be annotated with *invariants* that describe the routes each component may produce. Given the interfaces of a component's neighbors, we can then verify that the component respects its own interface. Modular reasoning can help operators abstract away unnecessary details, localize network bugs, confirm the validity of configuration updates, and scale verification to arbitrarily-large networks [Alberdingk Thijm et al. 2022]. When the interfaces imply some useful property such as reachability or access control, we can conclude that the system as a whole also satisfies that property.

We propose Timepiece, the first modular technique with *abstract network interfaces to verify a wide range of properties* (including route reachability). Kirigami [Alberdingk Thijm et al. 2022] proposed an architecture for modular control plane verification, but restricted its interfaces to only *exact* routes. Lightyear [Tang et al. 2022] presented an alternative verification technique with more expressive interfaces, but can only check that a network never receives a route (*e.g.*, for access control properties) — it cannot check reachability, a keen property of interest.

A temporal model. The basis of Timepiece's approach is a *temporal model of network execution*, where we reason over the states of nodes *at all times*. This model came as a surprise to us: one branch of prior work, starting with Minesweeper [Beckett et al. 2017a], sought to avoid the burden of reasoning over all transient states of the network by focusing on the *stable states* of the routing protocol once routing converges. Unfortunately, a naïve combination of modular reasoning and Minesweeper-style analysis of stable states *is unsound*. We discovered that the best way to recover soundness, while maintaining the system's generality, is to move to a temporal model.

This temporal model appears to ask the verification engine to do a lot more work: the system must verify that all the messages produced *at all times* are consistent with a user-supplied interface for each network component. Nevertheless, because reasoning is modular, ensuring individual problems are small, the system scales with the size of the largest *component* rather than the size of the network. This modular reasoning is general and any symbolic method (*e.g.*, symbolic simulation, model checking) could use it to verify individual components. We use a Satisfiability Modulo Theories (SMT)-based method in this work [Barrett and Tinelli 2018]. As a preview of modularity's benefits, Figure 1 shows the time it takes Timepiece to verify connectivity for variable-sized fattree topologies with external route announcements using the eBGP routing protocol, compared with a Minesweeper-style network-wide stable paths encoding.

Timepiece does require more work of users than monolithic, non-modular systems: users must supply interfaces that characterize the routes each network component may generate at each time. Still, the presence of these interfaces, once constructed, will have the typical benefits of interfaces in any software engineering context. First, they localize exactly where an error occurs: if a component is not consistent with its interface, then one must only search that component for the mistake, and a counterexample from the SMT solver can help pinpoint it. Second, router configurations change rapidly, and these changes are often the source of network-wide problems [Zhang et al. 2022]. Well-defined interfaces will be stable over time. As users update their configurations, they may easily recheck them against the stable, local interface for problems.

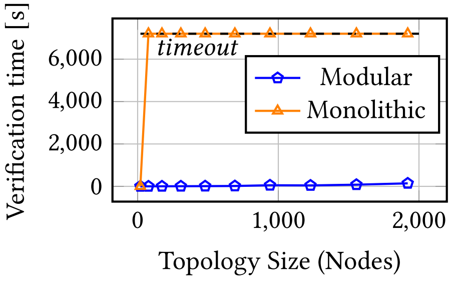


Fig. 1. Verification time comparison between Timepiece and Minesweeper-style verification.

Inspired by temporal logic [Pnueli 1984], we developed a simple language to help users specify their interfaces. Through this language, users may state that they expect to see certain sets of routes *always*, *eventually* (by some specified time t, to be more precise), or *until* (some approximate specified time). Moreover, the interface language allows users to generate abstract specifications that need not characterize irrelevant features of routes, and instead provide only what is necessary to prove a desired property. For instance, a user might specify a reachability property simply by stating a node must "eventually receive some route," without saying which route it must receive.

To summarize, the key contributions of this paper are:

- We demonstrate in depth why a natural, but naïve modular control plane analysis based on an analysis of stable states is unsound.
- We develop a new theory for modular control plane analysis based on time, and prove it sound and complete with respect to the network semantics. This theory is general, and can verify individual components using any verification method.
- We design and implement a new, modular control plane verification tool, Timepiece, based directly on this theory, which uses an SMT-based backend to reason symbolically about all possible routes at all times.
- We evaluate the tool and check a variety of policies at individual nodes in hundreds of milliseconds. Thanks to its embarrassingly parallel modular procedure, Timepiece scales to networks with thousands of nodes.

2 KEY IDEAS

This section introduces the stable routing model of network control planes, which serves as a foundation for many past network verification tools [Beckett et al. 2017a, 2018, 2019; Fogel et al. 2015]. It illustrates in depth why naïve adoption of this model for modular verification is unsound. It then introduces a new temporal model for control plane verification and provides the intuition for why the revised model is superior. This section is long but contains a substantial payoff: the essence of why a sound and general modular control plane analysis should be based off a temporal model of control plane behavior.

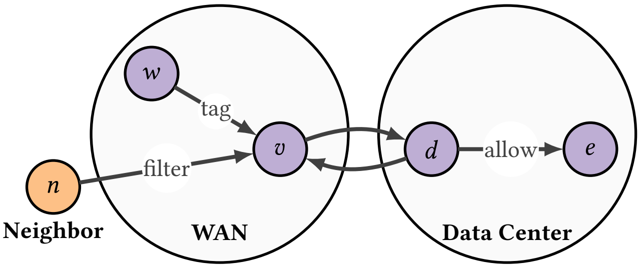
2.1 Background

To determine how to deliver traffic between two endpoints, routers (also called nodes) run distributed routing protocols such as BGP [Lougheed and Rekhter 1991], OSPF [Moy 1998], RIP [Hedrick 1988], or ISIS [Oran 1990]. Each node participating in a protocol receives *messages* (also called *routes*) from its neighboring nodes. After receiving routes from its neighbors, a node will select its “best” route—the route it will use to forward traffic. Different protocols use different metrics to compare routes and select the best among those received. For instance, RIP compares hop count; OSPF uses the shortest weighted-length path; and BGP uses a complex, user-configurable combination of metrics. Finally, each router sends its chosen route to its neighbors, possibly modifying the route along the way (for instance, by prepending its identifier to the path represented by the route).

Routing algebras. Routing algebras [Griffin and Sobrinho 2005; Sobrinho 2005] are abstract models that capture the similarities between different distributed routing protocols. Prior work on control plane verification [Beckett et al. 2017a; Giannarakis et al. 2020; Griffin et al. 2002] uses similar abstract models to formalize route computation. We adopt this standard abstract model of routing protocols, which specifies the following components.

- A directed graph G that defines the network topology’s nodes (V) and edges (E). We use lowercase letters (u, v, w, *etc.*) for nodes and pairs (uv) to indicate directed edges.
- A set S of *routes* that communicate routing information between nodes.
- An initialization function I that provides an initial route $I_v \in S$ for each node v.
- A set of edge transfer functions F. Each transfer function $f_e \in F$ transforms routes as they traverse the edge e.
- A binary function $\oplus$ (*a.k.a. merge* or the *selection function*) selects the best route between two options.

An idealized example. Many large cloud providers deploy data center networks to scale up their compute capacity. They connect those data centers to each other and the rest of the Internet via a wide-area network (WAN). To illustrate the challenges of modular network verification, we will explore verification of an idealized cloud provider network with WAN and data center components. The picture below presents a highly abstracted view of our network's topology.



The data center network contains routers d and e where d connects to the corporate WAN and e connects to data center servers. The WAN consists of routers w and v. Router v connects to the data center as well as to a neighboring network n, which is not controlled by our cloud provider.[1]

The default routing policy uses shortest-paths. However, in addition, the network administrators want e to be reachable from all cloud-provider-owned devices (*i.e.,* w, v, d), but not to be reachable from outsiders (*i.e.,* n). They intend to enforce this property by tagging all routes originating from their network (w) as "internal" (*e.g.,* using BGP community tags [CISCO 2005]) and allowing those routes to traverse the de edge. Doing so should allow e to communicate with internal machines but not external machines. Furthermore, to protect nodes from outside interference, the cloud provider applies route filters to external peers to drop erroneous advertised routes that may "hijack" [Feamster and Balakrishnan 2005] internal routing.

Modelling the example. To model our example network, we define the network topology as the graph G pictured earlier. We assume all routers participate in an idealized variant of eBGP [Lougheed and Rekhter 1991], which is commonly used in both wide-area networks and data centers [Abhashkumar et al. 2021]. The set of routes S used in this protocol are records with 3 fields: *(i)* an integer "local preference" that lets users overwrite default preferences, *(ii)* an integer path length, and *(iii)* a boolean tag field that is set to true if a route comes from an internal source and false otherwise. Lastly, S includes ∞, a message that indicates *absence* of a route.

Let's consider what happens when starting with a specific route at WAN node w, $\langle 100, 0, \text{false}\rangle$ (local preference 100, path length of 0, not tagged internal). The I function would assign w that route, and assign the ∞ route to all other nodes.

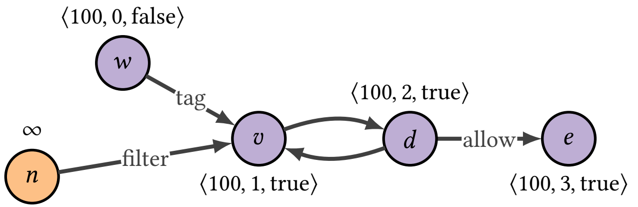
The transfer function f_e will increment the length field of every route by one across every edge e. In addition, edge wv sets the internal tag field to true and edge nv drops all routes (transforms them into ∞). Finally, edge de drops all routes not tagged internal/true.

The merge function $\oplus$ always prefers some route over the ∞ route. In addition, $\oplus$ prefers routes with higher local preference over lower local preference. If the local preference is the same, it chooses a route with a shorter path length. $\oplus$ ignores the tag field. For example, $\oplus$ operates as follows:

$$\begin{array}{ccccc}
\langle 100, 2, \text{false}\rangle & \oplus & \infty & = & \langle 100, 2, \text{false}\rangle \\
\langle 100, 2, \text{false}\rangle & \oplus & \langle 200, 5, \text{true}\rangle & = & \langle 200, 5, \text{true}\rangle \\
\langle 200, 2, \text{false}\rangle & \oplus & \langle 200, 5, \text{true}\rangle & = & \langle 200, 2, \text{false}\rangle
\end{array}$$

---

[1] Any of the edges could be bi-directional, allowing routes to pass in both directions, but for pedagogic reasons we strip down the example to the barest minimum, retaining edges that flow from left-to-right except for at v and d where routes may flow back and forth.



time	n	w	v	d	e
0	∞	⟨100, 0, false⟩	∞	∞	∞
1	∞	⟨100, 0, false⟩	⟨100, 1, true⟩	∞	∞
2	∞	⟨100, 0, false⟩	⟨100, 1, true⟩	⟨100, 2, true⟩	∞
3	∞	⟨100, 0, false⟩	⟨100, 1, true⟩	⟨100, 2, true⟩	⟨100, 3, true⟩
4	∞	⟨100, 0, false⟩	⟨100, 1, true⟩	⟨100, 2, true⟩	⟨100, 3, true⟩

Fig. 2. Simulation of the example network for a fixed set of initial routes. Node *e* learns a route to *w* since the route is tagged as internal, and the network stabilizes at time 3.

Network simulation. A *state* of a network is a mapping from nodes to the "best routes" they have computed so far. One may carry out a simulation by starting in the initial state and repeatedly computing new states (*i.e.,* new "best routes" for particular nodes). Eventually, well-behaved networks converge to *stable states* where no node can compute any better routes, given the routes provided by its neighbors.

To compute a new best route at a particular node, say v, we apply the f function to each best route computed so far at its neighbors w, n, and d, and then select the best route among the results and the initial value at v, using the merge (⊕) function. More precisely:

$$v_{new} = f_{wv}(w_{old}) \oplus f_{nv}(n_{old}) \oplus f_{dv}(d_{old}) \oplus I_v$$

The table in Figure 2 presents an example simulation. At each time step, all nodes compute their best route using the equation above, given the routes supplied by their neighbors at the previous time step. After time step 3, no node ever computes any new route—the system has reached a *stable state.* The picture in Figure 2 annotates each node in the diagram with the stable route it computes.[2]

Network verification. Since the edge from d to e only allows routes tagged internal, w's route would not reach e if v were to receive a better route from n (*e.g.,* if the route filter from n was implemented incorrectly). In other words, the simulation demonstrates that the network correctly operates when n sends no route (∞). But what about other routes? Will nv filter all routes from n correctly? SMT-based tools like Minesweeper [Beckett et al. 2017a] and Bagpipe [Weitz et al. 2016] can answer such questions by translating the routing problem into constraints for a Satisfiability Modulo Theory (SMT) solver to solve. In doing so, one may represent the set of all possible external route announcements symbolically and reason simultaneously about any external route announcement (something other verifiers such as Tiramisu [Abhashkumar et al. 2020], Plankton [Prabhu et al. 2017], Shapeshifter [Beckett et al. 2019], Hoyan [Ye et al. 2020], or DNA [Zhang et al. 2022] either do not do, or do partially).

---

[2] For simplicity, our model assumes a synchronous time semantics to simplify our examples and theory — however, we believe we could extend our approach to asynchronous time in the style of [Daggitt et al. 2018].

2.2 The Challenge of Modular Verification

A system for modular verification will partition a network into components and verify each component separately, possibly in parallel. However, since routes computed at a node in one component depend on the routes sent by nodes in neighboring components, each component must make some assumptions about the routes produced by its neighbors.

Interfaces. In our case, for simplicity (though this is not necessary), we place every node in its own component and define for it an *interface* that attempts to *overapproximate* the set of routes that the node might produce in a stable state. The interface for the network as a whole is a function A from nodes to sets of routes where $A(x)$ is the interface for node x.

The person attempting to verify the network will supply these interfaces. Of course, interfaces may be *wrong*—that is, they might not include some route computed by a simulation (and hence might not be a proper overapproximation). Indeed, when there are bugs in the network, the interfaces a user supplies are likely to be wrong! The user *expects* the network to behave one way, producing a certain set of routes, but the network behaves differently due to an error in its configuration. A sound modular verification procedure must detect such errors. On the other hand, a useful modular verification procedure should allow interfaces to overapproximate the routes produced, when users find it convenient. Overapproximations are sound for verifying properties over all routing behaviors of a network, and they often simplify reasoning, allowing users to think more abstractly.

Throughout the paper, we use predicates φ to define interfaces, where φ stands in for the set of routes $\{s \mid s \in S, \varphi(s)\}$. Returning to our running example, one might define the interface for w using the predicate $s.\text{lp} = 100 \land s.\text{len} = 0 \land \neg s.\text{tag}$. Such an interface would include exactly the one route generated by w in our example: $\langle 100, 0, \text{false}\rangle$. However, path length is unimportant in the current context; to avoid thinking about it, a user could instead provide a weaker interface representing infinitely many possible routes, such as $s.\text{lp} = 100 \land \neg s.\text{tag}$. This interface relieves the user of having to figure out the exact path length (not so hard in this simple example, but potentially challenging in an arbitrary wide-area network), and instead specifies only the local preference and the tag. In general, admitting overapproximations make it possible for users to ignore any features of routing that are not actually relevant for analyzing the properties of interest.

The Strawperson Verification Procedure. For a given node x, the *component centered at* x is the subgraph of the network that includes node x and all edges that end at x. Given a network interface A, our strawperson verification procedure (*SV*) will consider the component centered at each node x independently. Suppose a node x has neighbors $n_1, \ldots, n_k$. For that node x, *SV* checks that $\forall s_1 \in A(n_1), \ldots, \forall s_k \in A(n_k)$,

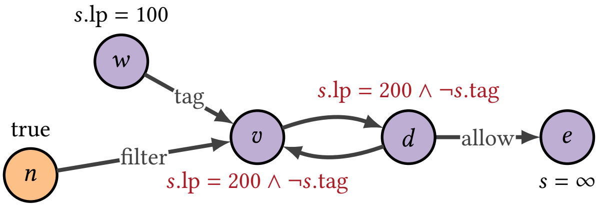
$$f_{n_1 x}(s_1) \oplus \cdots f_{n_k x}(s_k) \oplus I_x \in A(x) \tag{1}$$

This check is akin to performing one local step of simulation, checking that all possible inputs from neighbors give rise to an output route that conforms to the interface. One might *hope* that by performing such a check on *all* components independently, one would be able to guarantee that all nodes converge to stable states described by their interfaces. If that were the case, then one could verify properties by:

(1) Checking that all components guarantee their interfaces, under the assumption their neighbors do as well; and
(2) Checking that the interfaces imply the network property of interest (*e.g.,* reachability, access control, no transit).

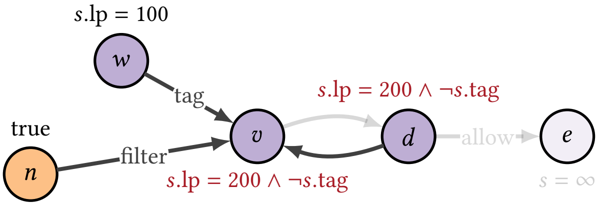
The problem: Execution interference. It turns out this simple and natural verification procedure is unsound: users can supply interfaces that, when analyzed in isolation, satisfy equation (1) above, but wind up excluding the stable states computed by simulation. Hence, the second verification step is pointless: a destination that appears reachable according to an interface may not be; conversely, a route that appears blocked may not be.

Let us reconsider the running example, where we assign w an initial route with local preference 100, and let us assume the external neighbor n can send us any route (true). A user could provide the interfaces shown below in order to falsely conclude that e will not receive a route from w.

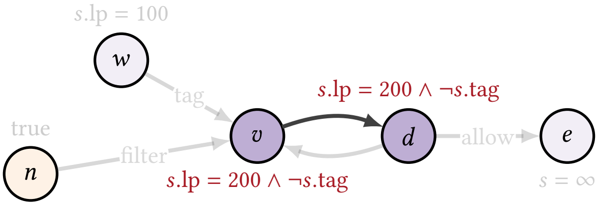


Here, it is easy to check that nodes n and w satisfy equation (1). Node n's interface is simply any route. Node w's route can be any route with a local preference of 100.[3]

The surprise comes at node v where its interface *only includes* routes that satisfy ¬s.tag, *i.e.*, routes not tagged as internal. Those routes have s.lp = 200 and may have any path length. But the route from w is tagged *true* along the edge wv — why is such a route erroneously excluded from v's interface? We show the component centered at v below. When computing its stable state, v will compare the routes it receives from w and d: because all routes from w have a local preference set to 100 by f_{wv}, whereas all routes from d have a better local preference of 200, v will always wind up selecting the route from d over the route from w.



But how then did d acquire these preferential routes tagged false? Such routes came in turn from v's interface. Here is the component centered at d.



What has happened is that v transmits its spurious routes to d, enabling d to justify its own spurious routes. d transmits these back again to v, where d's routes interact with the legitimate routes from w. Since w's routes have lower local preference, v discards them during computation of stable states. In a nutshell, the routes from an imaginary simulation proposed by our interface overrule

---

[3] It could be any route (true), as the edge wv applies the default preference of 100, but for clarity we label the routes at w with preference 100.

legitimate routes from the true simulation, through a process of circular self-justification at v and d. How might we prevent this *execution interference*?

Other approaches. There are a few ways to modify the suggested verification procedure to make it sound, but such modifications typically limit the power of the verification procedure or the expressiveness of the properties it can prove.

One approach is to limit every interface to exactly one route. Doing so avoids introducing any imaginary executions in the first place. Kirigami [Alberdingk Thijm et al. 2022] takes this approach, but the cost is that a network engineer analyzing their network must know *exactly* which routes appear at which locations. Computing routes exactly can be difficult in practice, and would seem unnecessary if all one cares about is a high-level property such as reachability. Moreover, it makes the interfaces brittle in the face of change—any change in network configuration likely necessitates a change in interface. A superior system would allow operators to define *durable* and *abstract* interfaces that imply key properties, and to check configuration updates against those interfaces.

Another approach is to limit the set of properties that the system can check to only those that say what *does not happen* in the network rather than what does happen. This is the approach Lightyear [Tang et al. 2022] takes. For instance, Lightyear allows one to check that node A will *not* be able to reach node B, but cannot prove that A and B will have connectivity — a common requirement in networks.

A final approach is to try to impose a static ordering on the components, and verify each component according to this ordering, using no information from the not-yet-verified components. This approach avoids the circular reasoning between d and e above, but is still unnecessarily conservative. The running example is overly simple as it shows routes propagated through a network in a single direction from left to right. In realistic networks, multiple destinations may broadcast routes in multiple directions at once. In such situations, there may be no way to order the components, and verification may not be possible. Moreover, we found that being unable to make assumptions from some neighbors made verifying certain properties, such as reachability, impossible in most cases.

2.3 The Solution: A Temporal Model

Our key insight is to change the model: rather than focus exclusively on the final stable states of a system, as a Minesweeper-style verifier would, we ensure that the model preserves the *entirety* of every step-by-step execution. To make this work, we need to add information to the model: a notion of *logical time*. By associating every route with the time at which a node computes it, we can *(i)* ensure that *all* routes at a particular time are properly considered, and their executions extended a time step, and *(ii)* ensure that we avoid collisions between routes computed at different times.

To verify such routing systems modularly, we once again must specify interfaces, but this time the interface for each node will specify the set of routes that may appear *at any time*. We write this now as an interface A that takes both a node *and a time* and returns an overapproximation of the set of routes that may appear at the node *at that time*. For example, $A(x)(t)$ now gives the set of routes for node x and time t. To check the interfaces, we use a verification procedure structured inductively with respect to time, as follows:

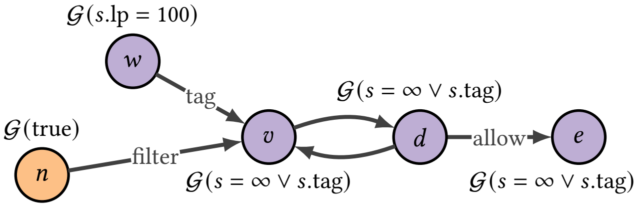
- At every node x, check I_x is included in $A(x)(0)$
- Consider each node x with neighbors $n_1, \ldots, n_k$. At time $t+1$, check that merging any combination of routes $s_1 \in A(n_1)(t), \ldots, s_k \in A(n_k)(t)$ from neighbors' interfaces at time t produces a route in $A(x)(t+1)$:

$$f_{n_1 x}(s_1) \oplus \cdots \oplus f_{n_k x}(s_k) \oplus I_x \in A(x)(t+1) \tag{2}$$

Because this procedure is structured inductively, we can prove, by induction on time, that all states at all times are included in their respective interfaces—the procedure is *sound*.

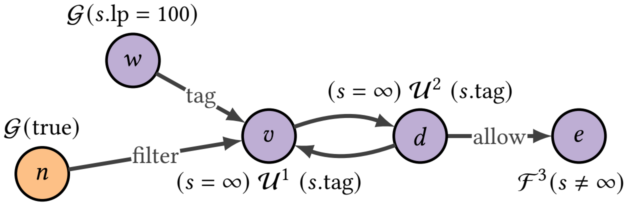
To reason over times, we borrow notation from temporal logic to specify interfaces. We write $\mathcal{G}(P)$ ("globally P") when a node's interface includes the routes that satisfy predicate P for all times t. We write $P_1 \; \mathcal{U}^t \; P_2$ ("P_1 until P_2") when a node may have routes satisfying P_1 until time $t-1$ and P_2 afterwards. Finally, we write $\mathcal{F}^t(P)$ ("finally P") to mean that eventually at time t routes start satisfying P.

Verifying correct interfaces. The picture below presents an interface we may verify with this model.


$\mathcal{G}(s.\text{lp} = 100)$
w
tag
$\mathcal{G}(s = \infty \vee s.\text{tag})$
$\mathcal{G}(\text{true})$
v
d
allow
e
n
filter
$\mathcal{G}(s = \infty \vee s.\text{tag})$
$\mathcal{G}(s = \infty \vee s.\text{tag})$

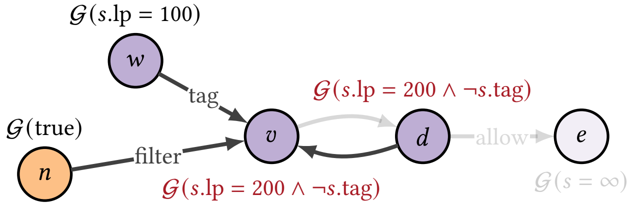
We once again assume that node n can send any route at any time, denoted by the interface: $\mathcal{G}(\text{true})$. Similarly, we assume w has some route with default local preference: $\mathcal{G}(s.\text{lp} = 100)$. The interesting part is at nodes v and d where the interfaces state that there is always either no route (*e.g.*, at the beginning of time), or a tagged route: $\mathcal{G}(s = \infty \vee s.\text{tag})$. Node e is now able to prove a weak property: *if it receives a route*, then the route will be tagged internal. Node v is able to prove its interface since routes are always tagged on import from node w, routes from n are correctly dropped, and any routes from d must also have a tag per its interface. In fact, all the nodes can prove their interface given their neighbors' interfaces.

Proving reachability. The previous interfaces were not strong enough to prove that w will be able to reach e. The problem is that we were trying to reason about *all* time, and yet e will not always have a route to w (*i.e.,* from time 0 onward). Instead, we know that e will *eventually* be able to reach w. Consider now the stronger interfaces shown below:


$\mathcal{G}(s.\text{lp} = 100)$
w
tag
$(s = \infty) \; \mathcal{U}^2 \; (s.\text{tag})$
$\mathcal{G}(\text{true})$
v
d
allow
e
n
filter
$(s = \infty) \; \mathcal{U}^1 \; (s.\text{tag})$
$\mathcal{F}^3(s \neq \infty)$

As before, we allow n and w to send any route. However, now nodes v and d declare that they will not have a route *until* a specified (logical) time, at which point they receive a tagged route. We give precise witness times for v and d's interfaces, as otherwise v could give d a non-null route (or vice-versa) that would violate the interface before its witness time. e's interface simply requires that e receives some route at the witness time (allowing arbitrary routes before the witness time). These interfaces are sufficient to prove that e will eventually receive a route to w, since d will eventually have a route tagged as internal, and hence e will allow it.

Debugging erroneous interfaces. Let us revisit the example from before where the user provided unsound interfaces by introducing spurious routes with local preference 200. The equivalent interfaces are now shown below.


$\mathcal{G}(s.\text{lp} = 100)$
w
tag
$\mathcal{G}(s.\text{lp} = 200 \land \neg s.\text{tag})$
$\mathcal{G}(\text{true})$
v
d
allow
e
filter
n
$\mathcal{G}(s.\text{lp} = 200 \land \neg s.\text{tag})$
$\mathcal{G}(s = \infty)$

Unlike before, the verification procedure detects an error: the interfaces at nodes v and d do not include the initial route ∞ at time 0. As a result, the user will receive a counterexample for time $t = 0$ when verifying v or d. Suppose our imaginative user tries to circumvent this issue by also including the initial route in the interfaces for v and d with the interface:

$$\mathcal{G}\big((s.\text{lp} = 200 \land \neg s.\text{tag}) \lor (s = \infty)\big)$$

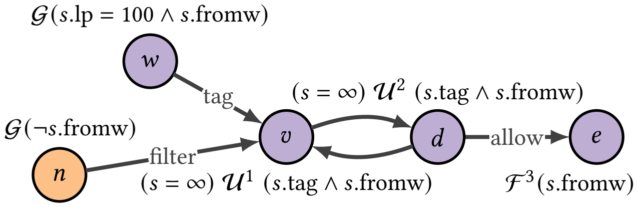
However, doing so merely pushes the problem "one step forward in time"—there is no way to circumvent our temporal analysis. If d's route may be ∞, v's interface must also consider what routes it selects when that is the case, including tagged routes such as $\langle 100, 1, \text{true}\rangle$. The user might receive a counterexample at time $t = 1$ where v's route is the following:

$$f_{wv}(\langle 100, 0, \text{false}\rangle) \oplus f_{nv}(\infty) \oplus f_{dv}(\infty) = \langle 100, 1, \text{true}\rangle$$

where $A(v)(1)$ does not contain the result $\langle 100, 1, \text{true}\rangle$. This counterexample should reveal the fact that there is an error in either the specification (as in this case) or the configuration (*e.g.*, if a buggy configuration tagged routes from w false rather than true as expected).

Properties and ghost state. Since modular properties can only reference the route at any single node, an observant reader may wonder if this limits the kinds of properties that we can verify. In the examples so far, we checked to see whether or not node e receives *some* route, but did not guarantee that the route originated at w. Fortunately, there is a simple fix.

Enter ghost state, a technique used in a wide variety of settings (see Dafny [Leino 2010], for instance). Users may model routes as containing additional "ghost" fields that play no role in a protocol's routing behavior, yet can capture end-to-end properties. For instance, we could add an additional "ghost" field to determine if a route initiated at node w—let us call that field "fromw." We assume this field is initially true at w, false at all other nodes, and that transfer functions preserve this field. With this addition, we can now check that e receives a route from w and no other node:


$\mathcal{G}(s.\text{lp} = 100 \land s.\text{fromw})$
w
tag
$(s = \infty)\ \mathcal{U}^2\ (s.\text{tag} \land s.\text{fromw})$
$\mathcal{G}(\neg s.\text{fromw})$
v
d
allow
e
filter
n
$(s = \infty)\ \mathcal{U}^1\ (s.\text{tag} \land s.\text{fromw})$
$\mathcal{F}^3(s.\text{fromw})$

Ghost state allows us to specify and check many network properties; Table 1 presents a variety of other possibilities. That said, while ghost state is general and flexible, it can only capture information about the history of a route at a *single* node and is thus not a panacea. For instance, properties involving the routes at more than one node, such as a formulation of local equivalence [Beckett et al. 2017a], where $\sigma(u)(t) = \sigma(v)(t)$ for some arbitrary u, v and t, is inexpressible using our verifier.

3 FORMAL MODEL WITH TEMPORAL INVARIANTS

Figure 3 provides a summary of the key definitions and notation needed to formalize our verification procedure. The notation follows from the previous section, *e.g.*, a network instance $N = (G, S, I, F, \oplus)$

Table 1. Ghost state for selected example properties.

Property	**Added ghost state**
reachability to d [Fogel et al. 2015]	*1 bit to mark routes from d*
isolation [Beckett et al. 2017a]	*1 bit per isolation domain*
ordered waypoint [Kazemian et al. 2013].	$log_2(k)$ *bits for* k *waypoints*
unordered waypoint [Beckett et al. 2017a]	k *bits for* k *waypoints*
no-transit [Beckett et al. 2016]	*mark with* {peer, prov, cust}
fault tolerance [Beckett et al. 2017a]	*up to* $\|E\|$ *bits*
bounded path length [Lopes et al. 2015]	*integer length field*

Network instances $\boxed{N = (G, S, I, F, \oplus)}$

$G = (V, E)$	network topology
V	topology nodes
$E \subseteq V \times V$	topology edges
S	set of network routes
$s \in S$	a route
$I : V \to S$	node initialization function
$I_v \in S$	initial route at node v
$F : E \to (S \to S)$	edge transfer functions
$f_e : S \to S$	transfer function for edge e
$\oplus : S \times S \to S$	merge function

Network semantics $\boxed{\sigma : V \to \mathbb{N} \to S}$

$\sigma(v)(t) \in S$	state at node v at time t
$preds(v) = \{u \mid u \in V, uv \in E\}$	in-neighbors of v

$$\sigma(v)(0) = I_v \tag{3}$$

$$\sigma(v)(t+1) = I_v \oplus \bigoplus_{u \in preds(v)} f_{uv}(\sigma(u)(t)) \tag{4}$$

Fig. 3. Summary of our formal routing model and notation.

contains the key components introduced earlier. To refer to the route computed by a network simulator at node v at time t, we use the notation $\sigma(v)(t)$ (defined as before—see Figure 3).

Figure 4 presents our interfaces and language of temporal operators with lifted versions of some common set operations. As before, we use A to denote network interfaces. A valid interface is an *inductive invariant* [Giannakopoulou et al. 2018]. Such interfaces satisfy the *initial and inductive conditions* specified in Figure 4. Valid interfaces may be used to prove node properties, as specified by the *safety condition* in Figure 4.

The most important property of our system is *soundness*: the simulation states are included in any interface A that satisfies the initial and inductive conditions.

THEOREM 3.1 (SOUNDNESS). *Let A satisfy initial and inductive conditions. Then $\forall t \in \mathbb{N}, \forall v \in V, \sigma(v)(t) \in A(v)(t)$.*

PROOF. By induction on t. See A.1. □

Interfaces and Properties

$$\begin{array}{lll} A & : V \to \mathbb{N} \to 2^S & \text{node interfaces/invariants} \\ P & : V \to \mathbb{N} \to 2^S & \text{node properties} \\ \varphi & : 2^S & \text{sets of states} \end{array}$$

Temporal operators $\boxed{Q : \mathbb{N} \to 2^S}$

$$\begin{array}{lcll} \mathcal{G}\varphi & = & \lambda\, t.\, \varphi & \text{globally} \\ \varphi_1\ \mathcal{U}^\tau\ \varphi_2 & = & \lambda\, t.\, \mathit{if}\ t < \tau\ \mathit{then}\ \varphi_1 \mathit{else}\ \varphi_2 & \text{until} \\ \mathcal{F}^\tau\varphi & = & \lambda\, t.\, S\ \mathcal{U}^\tau\ \varphi & \text{finally} \\ Q_1 \sqcap Q_2 & = & \lambda\, t.\, Q_1(t) \cap Q_2(t) & \text{intersection (lifted)} \\ Q_1 \sqcup Q_2 & = & \lambda\, t.\, Q_1(t) \cup Q_2(t) & \text{union (lifted)} \\ {\sim}Q & = & \lambda\, t.\, S \setminus Q(t) & \text{negation (lifted)} \end{array}$$

Verification Conditions

Initial condition for A:

$$\forall v \in V,\ I_v \in A(v)(0) \tag{5}$$

Inductive condition for A:

$$\begin{gathered} \forall v \in V, u_1, u_2, \ldots, u_n \in \mathit{preds}(v), \forall t \in \mathbb{N}, \\ \forall s_1 \in A(u_1)(t), \forall s_2 \in A(u_2)(t), \ldots, \forall s_n \in A(u_n)(t), \\ \left(I_v \oplus \bigoplus_{i \in \{1,..,n\}} f_{u_i v}(s_i) \right) \in A(v)(t+1) \end{gathered} \tag{6}$$

Safety condition for P with respect to A:

$$\forall v \in V, \forall t \in \mathbb{N}, A(v)(t) \subseteq P(v)(t) \tag{7}$$

Fig. 4. Summary of our interfaces and properties, temporal operators and verification conditions.

Since initial and inductive conditions suffice to prove that simulation states are included within interfaces, it is safe in turn to use interfaces to check node properties.

Corollary 3.2 (Safety). *Let A satisfy initial and inductive conditions. Let P satisfy the safety condition with respect to A. Then $\forall t \in \mathbb{N}, \forall v \in V, \sigma(v)(t) \in P(v)(t)$.*

Proof. From definitions. See A.2. □

Our verification procedure is also *complete* in the sense that for any network, there exists an interface that characterizes its behavior exactly. One of the consequences of completeness is that our modular verification procedure is powerful enough to prove any property that we could prove via simulation.

Theorem 3.3 (Completeness). *Let σ be the state of the network. Then $A(v)(t) = \{\sigma(v)(t)\}$ for all $v \in V$ and all $t \in \mathbb{N}$ satisfies the initial and inductive conditions.*

Proof. By construction of the interface. See A.3. □

4 SMT ALGORITHMS FOR VERIFICATION

Each of the verification conditions (initial, inductive, safety) universally quantify over the (finitely many) nodes in the network. Consequently, a modular verifier can enumerate the network's nodes and independently check each VC for a concrete choice of node. To check an instance of a VC, an off-the-shelf SMT solver will attempt to prove the condition is valid (*i.e.*, true for all choices of t). If the instance is *not valid*, the solver can provide us with a *counterexample*—the state of a node at a particular time that the VC does not hold. Counterexamples to initial or inductive conditions indicate that the interface does not approximate the network's behavior, while a counterexample to the safety condition indicates that the interface is not strong enough to prove the property. The latter case may occur because the property is simply not true (indicating a bug), or alternatively because we must strengthen the given interface in order to prove the property.

To differentiate our modular procedure from prior work, we present a Minesweeper-style [Beckett et al. 2017a] monolithic checking procedure in Algorithm 1. This algorithm does not refer to time, but instead encodes the *stable states* of the network as a single formula ψ (as presented in §2.1). Given a property P over the stable states (note that here, P also ignores time), it checks if P always holds given these states by calling IsValid to ask the solver if $\psi \to P$ is always true.

Algorithm 1 Minesweeper-style checking algorithm.

```
proc CheckMono(network $(G, S, I, F, \oplus)$, property $P$)
    $\psi \leftarrow$ EncodeStableStates($G$)
    return IsValid($\psi \to P$)
```

We present our modular checking procedure in Algorithm 2. The outer CheckMod function iterates over each node of the network and encodes the underlying formula (for the current node) of our three verification conditions by calling EncodeInitCond, EncodeIndCond and EncodeSafeCond ((5), (6) and (7), respectively). As in Algorithm 1, we then ask the solver if every encoded formula is valid using IsValid. If IsValid returns false for any check, we then can ask for the relevant counterexample model m.

Algorithm 2 The modular checking algorithm.

```
proc CheckMod(network $(G, S, I, F, \oplus)$, interface $A$, property $P$)
    for all $v \in V$ do in parallel
        $\psi_1 \leftarrow$ EncodeInitCond($v$)                ▷ (5)
        $\psi_2 \leftarrow$ EncodeIndCond($v$)                 ▷ (6)
        $\psi_3 \leftarrow$ EncodeSafeCond($v$)                ▷ (7)
        if $\bigvee_{1 \leq i \leq 3} \neg$IsValid($\psi_i$) then return false
    return true
```

Importantly, unlike Algorithm 1, by using temporal invariants we are able to separate the task of checking P on the network instance into three independent verification tasks, with the encoding of initial and inductive conditions roughly proportional in size to the complexity of the policy at the given node (which in turn is related to the in-degree of the node—denser networks that include nodes with higher in-degree are more expensive to check). The encoding of the safety condition is proportional to the size of the formulae describing the interface and property (and is generally tiny). In addition to reducing the size of each SMT formula, the factoring of the problem into independent conditions makes it possible to use parallelism to check conditions on nodes simultaneously. We will discuss the performance implications of our procedure further in §6.

If we get back a counterexample m, we can inspect it to understand why our checks failed. This can provide insight into how to strengthen the invariants, or pinpoint a bug in our policy. Anecdotally, this feature was critical to helping us design interface functions for our own experiments, and for discovering bugs in our modelling of network policies.

Table 2. Lines of C# code to write the properties and interfaces for each benchmark.

Benchmark	Property LoC	Interface LoC
Reach	2	3
Len	5	7
Vf	2	12
Hijack	4	21
BlockToExternal	5	5

5 IMPLEMENTATION

We implemented TIMEPIECE's modular verification procedure as a library written in C#. The library allows users to construct models of networks and then modularly verify them. Like the network modelling framework NV [Giannarakis et al. 2020], TIMEPIECE allows users to customize their models by choosing the kinds of routes (which may involve integers, strings, booleans, bitvectors, records, optional data, lists, or sets) and the way initialization, transfer and merge functions process them. This modelling language makes it easy to add ghost state to routes, as described earlier. It is also possible to declare and use symbolic values in the model. Hence, one may reason about all possible prefixes or more generally about all possible external routing announcements.

For example, to model a network running eBGP, we would adopt many of the modelling choices made in Minesweeper [Beckett et al. 2017a]. For instance, we use integers and the SMT theory of integers to model path length. We use bitvectors to model local preference and MED. We use the theory of arrays to model sets of community tags.

Under the hood, TIMEPIECE uses Microsoft's Zen verification library [Beckett and Mahajan 2020], which in turn serves as an interface to the Z3 SMT solver. Hence, the only practical limits to a user's network model are those that arise from the features and theories supported by Z3.

TIMEPIECE uses multi-threading to run modular checks in parallel.[4] As each check is independent, the time to set up additional threads is the only overhead for parallelization.

6 EVALUATION

To evaluate TIMEPIECE and illustrate its scaling trends, we generated a series of synthetic fattree [Al-Fares et al. 2008] data center networks and verified four variations on reachability properties. We also verified an isolation property on a real wide-area network configuration with over 100,000 lines of code. Table 2 shows how the number of lines of code needed to write the interfaces for each of our benchmarked properties is low-effort. We generated interfaces for our experiments parametrically for any size of network, based on the distinct *roles* of nodes: for fattree networks, a node's pod and tier determined its role (5 roles, discussed below); for our wide-area network benchmark, we distinguished internal nodes from external neighbors (2 roles).

To compare our implementation against a baseline, we implemented the monolithic, network-wide Minesweeper-style procedure Ms from Algorithm 1 and compared its performance against TIMEPIECE. Ms analyzes stable states, which are independent of time. To compare Ms with TIMEPIECE, we first crafted properties for TIMEPIECE, which employs timed invariants. We erased the temporal components of these invariants to generate properties that Ms could manage. For instance, when TIMEPIECE would verify properties of the form $\mathcal{G}\varphi$, $\mathcal{F}^t\varphi$, or $\varphi_2\ \mathcal{U}^t\ \varphi$, Ms would instead verify that the network's stable states satisfy φ.

---

[4] We use C#'s Parallel LINQ library [Microsoft 2021], which can run up to 512 concurrent threads.

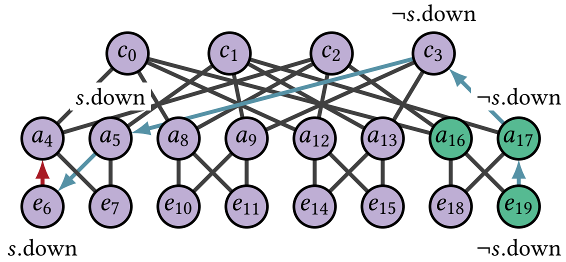


Fig. 5. An example fattree network, showing how Vf sets s.down along the path between the destination node e_{19} and e_6. e_6a_4 will drop the route from e_6 to prevent valley routing.

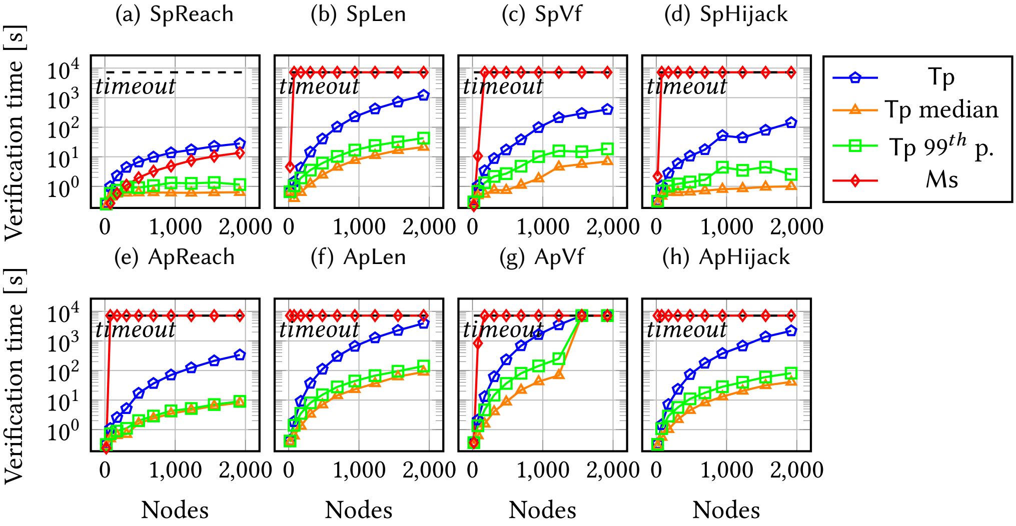


Fig. 6. Ms *vs.* Tp verification times for fattree benchmarks with 8 different policies.

We ran all our benchmarks on a Microsoft Azure D96s v5 virtual machine with 96 vCPUs and 384GB of RAM. We took advantage of the machine's multi-core processor to run all the modular checks in parallel, while monolithic checks necessarily ran on a single thread. We timed out any benchmark that did not complete in 2 hours. We reported four times for each of our benchmarks: *(i)* the total time until all Timepiece threads finished (Tp); *(ii)* the median node check time; *(iii)* the 99th percentile node check time (99% of checks completed in less than this much time); and *(iv)* the total time taken by Ms.

Fattrees. We parameterize our fattree networks by their number of pods k: a k-fattree has $1.25k^2$ nodes and k^3 edges: Figure 5 shows an example 4-fattree as part of our Vf policy. We considered multiples of 4 for $4 \leq k \leq 40$ to assess Timepiece's scalability: whereas we expected a monolithic verifier to time out on larger topologies, we hypothesized that Timepiece would scale to these networks. We present how verification time grows with respect to the number of nodes in each fattree in Figure 6. Note that the figure shows verification time on a logarithmic scale.

Table 3. eBGP route fields modelled by TIMEPIECE in SMT for fattree benchmarks.

Route field	Modelled type in SMT
Route destination	bitvector [SMT-LIB 2010b]
Administrative distance	bitvector [SMT-LIB 2010b]
eBGP local preference	bitvector [SMT-LIB 2010b]
eBGP multi-exit discriminator	bitvector [SMT-LIB 2010b]
eBGP origin type	{egp, igp, unknown} [SMT-LIB 2010b]
eBGP AS path length	integer [SMT-LIB 2010c]
eBGP communities	set<string> [SMT-LIB 2010a, 2020]

We considered four different properties: reachability (Reach), bounded path length (Len), valley freedom (Vf) and route filtering (Hijack). We tested each property for routing to a fixed destination edge node (Sp), and all-pairs routing to any edge node (Ap). Our routes modelled the eBGP protocol in these networks. Table 3 summarizes the eBGP fields represented and how we modelled them in SMT. We model the major common elements of eBGP routing: a route destination as a 32-bit integer (representing an IPv4 prefix); administrative distance, local preference, multi-exit discriminators as 32-bit integers (encoded as bitvectors); eBGP origin type as a ternary value; the AS path length as an (unbounded) integer; and BGP communities as a set of strings.

Witness times. For our fattree properties, we wanted to prove that nodes *eventually* received routes to the destination node. To establish that a node v eventually has a route at time t, we must show that one of v's neighbors sent it a route at an earlier time $t-1$. The destination node *dest* will have a route at time 0, its neighbors will have routes at time 1, and so on for their neighbors at time 2. In the Sp case, *dest* is a concrete node; in the Ap case, we used a symbolic *dest* to consider any choice of edge node as the destination. A node v will receive a route to *dest* at different times depending on where *dest* is relative to v. This breaks down to five cases (or *roles*), following the fattree's structure, depending on whether v is *(i)* the *dest* node (0 hops, $t = 0$); *(ii)* an aggregation node in *dest*'s pod (1 hop, $t = 1$); *(iii)* a core node, or an edge node in *dest*'s pod (2 hops, $t = 2$); *(iv)* an aggregation node in another pod ($t = 3$); or *(v)* an edge node in another pod ($t = 4$). These cases mirror those identified for local data center invariants in [Jayaraman et al. 2019a]. For brevity, we use a function $dist(v)$ to specify these times for a node v.

Reach. Reach demonstrates the simplest possible routing behavior and serves as a useful baseline. The policy simply increments the path length of a route on transfer. We initialized one destination edge node with a route to itself, and all other nodes with no route (∞). Our goal is to prove every node eventually has a route to the destination (*i.e.*, its route is not ∞). More precisely, because our network has diameter 4, each node v should acquire a route in 4 time steps.

$$P_{\text{Reach}}(v) \equiv \mathcal{F}^{4}(s \neq \infty)$$

Interfaces for these benchmarks mirror the simplicity of the policy and property. If a node's route $s \neq \infty$ at time t, then its neighbors will in turn have a route $s \neq \infty$ at time $t+1$.

$$A_{\text{Reach}}(v) \equiv \mathcal{F}^{dist(v)}(s \neq \infty)$$

SpReach's policy and property are so simple that Tp is actually slightly *slower* than Ms, as shown in Figure 6a. We conjecture the Ms encoding reduces to a particularly easy SAT instance. That said, we can already see that individual checks in TIMEPIECE take only a fraction of the time that Ms takes, with 99% of node checks completing in at most 1.1 seconds, even for our largest benchmarks.

For ApReach, Figure 6e shows that, perhaps from the burden of modelling the symbolic *dest*, Ms times out at $k = 8$. Tp verifies our largest benchmark ($k = 40$, with 2,000 nodes) in 5.5 minutes, with 99% of individual node checks taking under 9 seconds.

Len. Our next benchmark uses the same policy as Reach, but considers a stronger property: every node eventually has a route of at most 4 hops to the destination.

$$P_{\mathsf{Len}}(v) \equiv \mathcal{F}^{4}\big(s.\mathrm{len} \leq 4\big)$$

To prove this property, our interfaces specify that path lengths in routes should not exceed the distance to the destination: $s.\mathrm{len} \leq \mathit{dist}(v)$. In addition, because local preference influences routing, we fix the local preference to the default for all routes when present: $s.\mathrm{lp} = 100$.

$$A_{\mathsf{Len}}(v) \equiv \underbrace{\mathcal{G}\big(s = \infty \lor s.\mathrm{lp} = 100\big)}_{\textit{no better routes appear}} \sqcap \underbrace{\mathcal{F}^{\mathit{dist}(v)}\big(s.\mathrm{len} \leq \mathit{dist}(v)\big)}_{\textit{eventually the route appears}}$$

Reasoning over path lengths requires Z3 to use slower bitvector and integer theories. Figure 6b shows that monolithic verification times out at $k = 12$ for SpLen. By contrast, modular verification is able to solve $k = 40$ in just over 20 minutes, with 99% of nodes verified in under 43 seconds. Figure 6f shows that monolithic verification is not even possible for ApLen at $k = 4$; Tp completes for ApLen $k = 40$ in around 66 minutes, with 99% of nodes verified in 2.4 minutes.

Vf. Vf extends Reach with policy to prevent up-down-up (valley) routing [Beckett et al. 2016, 2017b; Pepelnjak 2018], where routes transit an intermediate pod. To implement this policy, we add a BGP community D along "down" edges in the topology (*i.e.,* from a core node or from an aggregation node to an edge node), and drop routes with D on "up" edges (see Figure 5). For brevity, we write "$s.\mathrm{down}$" to mean "$D \in s.\mathrm{tags}$". We test the same reachability property as Reach.

The legitimate routes in the fattree all start as routes travelling up from the destination node's pod, *e.g.,* the nodes in green (a_{16}, a_{17}, e_{19}) in Figure 5. We refer to these nodes as "adjacent nodes" with a shorthand $\mathit{adj}(v)$: they transmit routes to the core nodes (and thereby to the rest of the network) along their up edges. These edges will drop the routes if $s.\mathrm{down}$, so we require that $\mathit{adj}(v) \longrightarrow \neg s.\mathrm{down}$. To ensure this, we add conjuncts to our interfaces requiring that nodes' final routes are no better than the shortest path's route: $s.\mathrm{lp} = 100 \land s.\mathrm{len} = \mathit{dist}(v)$. This ensures our inductive condition holds after every node has a route: otherwise, a core node (for instance) could offer a spurious route with $s.\mathrm{len} < 1 \land s.\mathrm{down}$ to an adjacent node.

$$A_{\mathsf{Vf}}(v) \equiv \big(s = \infty\big)\ \mathcal{U}^{\mathit{dist}(v)}\ \Big(\underbrace{s.\mathrm{lp}{=}100 \land s.\mathrm{len}{=}\mathit{dist}(v)}_{\textit{no better routes appear}} \land \underbrace{\big(\mathit{adj}(v) \longrightarrow \neg s.\mathrm{down}\big)}_{\textit{adjacent nodes will share routes}}\Big)$$

Figure 6c shows that Ms verifies up to $k = 8$ before timing out. As with SpLen, Tp time grows gradually in proportion to the number of nodes, topping out at 6.6 minutes for $k = 40$, with all node checks completing in under 20 seconds. Figure 6g shows that for all-pairs routing, monolithic verification times out again at $k = 12$, whereas Tp hits the 2-hour timeout at $k = 36$. We conjecture this may be due to the added complexity of encoding $\mathit{adj}(v)$ when the destination is symbolic.

Hijack. Hijack models a fattree with an additional "hijacker" node h connected to the core nodes. h represents a connection to the Internet from outside the network, which may advertise *any* route. We add a boolean ghost state tag to S for this policy to mark routes as external (from h) or internal. The destination node will advertise a route with $s.\mathrm{prefix} = p$, where p is a symbolic value representing an internal address: the core nodes will then drop any routes from h for prefix p, but allow other routes through. Apart from this filtering, routing functions as in the Reach benchmarks.

For this network, we verified that every internal node eventually has a route for prefix p and which is not via the hijacker ($\neg s.\text{tag}$), assuming nothing about the hijacker's route ($A_{\text{Hijack}}(h) \equiv \mathcal{G}(\text{true})$).

$$P_{\text{Hijack}}(v) \equiv \mathcal{F}^{4}(s.\text{prefix}{=}p \land \neg s.\text{tag})$$

The Hijack interface is straightforward. We must simply re-affirm that nodes with internal prefixes never have external routes: $s.\text{prefix} = p \rightarrow \neg s.\text{tag}$. Once nodes have received a route from the destination at time $dist(v)$, they should keep that route forever, and hence their route will have both $s.\text{prefix} = p$ and $\neg s.\text{tag}$.

$$A_{\text{Hijack}}(v) \equiv \underbrace{\mathcal{F}^{dist(v)}(s.\text{prefix}{=}p \land \neg s.\text{tag})}_{\textit{route will be internally reachable}} \sqcap \underbrace{\mathcal{G}(s.\text{prefix}{=}p \rightarrow \neg s.\text{tag})}_{\textit{no hijack route is ever used}}$$

In SpHijack, monolithic verification times out at $k = 8$, whereas modular verification time scales to $k = 40$. 99% of nodes complete their checks in under 3 seconds, with our longest check taking 10.7 seconds at $k = 40$. As with our other benchmarks, verification time grows as the in-degree of each node — which determines the size of the SMT encoding of our inductive condition — grows linearly with respect to k. Figure 6h shows similar patterns for the all-pairs case, with *no* monolithic benchmark completing on time, and modular verification taking at most 36.6 minutes.

Wide-area networks. To better investigate TIMEPIECE's scalability for other types of networks, we evaluated it on the Internet2 [Internet2 2013] wide-area network.[5] We converted the configuration files to TIMEPIECE's model by extracting the policy details using Batfish [Fogel et al. 2015]. The resulting network has 10 internal nodes within Internet2's AS and 253 external neighbors. We did not model all components of Internet2's routing policies: we focused on IPv4 and BGP routing, and treated some complex behaviors as "havoc" (soundly overapproximating the true behavior).[6] We do not know Internet2's intended routing behavior: because of this, we cannot be certain that a counterexample found by TIMEPIECE represents a real violation of the network's behavior; nonetheless, we may still use this network to assess how well TIMEPIECE enables modular verification.

It appears that Internet2 uses a **BTE** community tag to identify routes that must not be shared with external neighbors. We checked that, if the internal nodes initially have any possible route, then no external neighbor of Internet2 should ever obtain a route with the **BTE** tag set, assuming the external neighbors do not start with such routes.

$$P_{\text{BlockToExternal}}(v) \equiv \begin{cases} \mathcal{G}(s \neq \infty \rightarrow \textbf{BTE} \notin s.\text{tags}) & \text{if } v \text{ is external} \\ \mathcal{G}(\text{true}) & \text{otherwise} \end{cases}$$

We used the property directly as our interface, *i.e.*, $\forall v.\ A_{\text{BlockToExternal}}(v) \equiv P_{\text{BlockToExternal}}(v)$. Modular checking remains fast despite the network's more complex policies: on a 6-core Macbook Pro with 16GB of RAM, modular verification completes in 38.3 seconds, with a median check time of 0.6 seconds and a 99th percentile check time of 4.2 seconds. Monolithic verification does not complete after 2 hours.

7 RELATED WORK

Our work is most closely related to other efforts in control plane verification [Abhashkumar et al. 2020; Alberdingk Thijm et al. 2022; Beckett et al. 2017a, 2018, 2019; Fogel et al. 2015; Gember-Jacobson et al. 2016; Giannarakis et al. 2020; Lopes and Rybalchenko 2019; Prabhu et al. 2017; Tang et al. 2022; Weitz et al. 2016; Ye et al. 2020]. The following paragraphs recommend different classes of tools depending on the specifics of one's verification problem.

---

[5] We used the versions of Internet2's configuration files available here [Weitz 2016].

[6] These include prefix matching, community regex matching and AS path matching. We also did not model BGP nexthop.

If your network is small (in the tens of nodes), then we recommend an SMT-based tool such as Minesweeper [Beckett et al. 2017a] or Bagpipe [Weitz et al. 2016] for their ease-of-use, generality, and symbolic reasoning. Minesweeper supports a broad range of properties including reachability, waypointing, no blackholes and loops, and device equivalence.

If your network is larger, and neither incremental recomputation after device update nor fully symbolic reasoning is important, use a simulation-based tool [Beckett et al. 2019; Fogel et al. 2015; Giannarakis et al. 2020; Lopes and Rybalchenko 2019; Prabhu et al. 2017; Ye et al. 2020]. Some of these tools also employ symbolic reasoning in limited ways to provide useful capabilities. For example, inspired by effective work on data plane analysis [Khurshid et al. 2013], Plankton [Prabhu et al. 2017] first analyzes configurations to identify IP prefix equivalence classes. Identified equivalence classes may be treated symbolically in the rest of the computation. Plankton might be able to reason symbolically about other attributes, but doing so would require additional custom engineering to find the appropriate sort of equivalence class ahead of time (*e.g.,* for BGP AS paths or communities). With Timepiece, any component may be treated symbolically and the solver effort is passed off to the underlying SMT engine.

If your network is large and symbolic reasoning is important, then there are fewer options to consider. Bonsai exploits symmetry to derive smaller abstractions of a network [Beckett et al. 2018], but does not work if the network has topology or policy asymmetries or one considers failures (which break topological symmetries). Other tools exploit modularity in network designs. Kirigami [Alberdingk Thijm et al. 2022] applies assume-guarantee reasoning for control plane verification, but requires interfaces to specify the exact routes passed between any two components. As such, it is impossible to craft interfaces that are robust to minor changes in network policies. Lightyear [Tang et al. 2022] allows users to craft more general interfaces, but can only prove a limited range of properties, which do not include reachability properties. We conjecture (but have not proven) that Lightyear verifies properties that Timepiece would express as $\mathcal{G}\varphi$, but not properties that require $\mathcal{U}^t$ or $\mathcal{F}^t$ temporal operators. On the other hand, Lightyear checks one neighbor at a time rather than all neighbors at once, and hence may be more efficient.

Daggitt *et al.* also use a timed model [Daggitt et al. 2018]. However, our verification method and target properties differ — they focus on convergence properties of routing protocols, whereas we analyze properties that depend upon a network's topology and configuration such as reachability.

Other inspirations for our work are SecGuru and RCDC [Jayaraman et al. 2019a] in data plane verification. Unlike our work, they use non-temporal invariants, which they extract from the network topology and assume as ground truth for policies on individual devices. Whereas our experiments likewise used the topology to define local invariants, our verification procedure checks that these invariants are in fact guaranteed by the other devices in the network.

Compositional reasoning. Our work is inspired by the success of automated methods for compositional verification of concurrent systems – a recent handbook chapter [Giannakopoulou et al. 2018] provides many useful pointers to the rich literature on this topic. Automated methods using compositional reasoning have been successfully applied in many application domains – concurrent programs (*e.g.,* [Flanagan and Qadeer 2003; Gupta et al. 2011; Owicki and Gries 1976]), hardware designs (*e.g.,* [Henzinger et al. 2000; Kurshan 1988; McMillan 1997], reactive systems (*e.g.,* [Alur and Henzinger 1999]) — and for a range of properties including safety and liveness, as well as for refinement checking. [Lomuscio et al. 2010] applies assume-guarantee reasoning for verifying stability of network congestion control systems. Many such applications use temporal logic for specifying properties, as well as assumptions and guarantees at component interfaces [Pnueli 1984]. One main challenge is to come up with suitable assumptions that are strong enough to prove the properties of interest. Toward this goal, Timepiece uses a language of temporal invariants inspired by temporal logic to support checking local (*i.e.,* per-router) properties. However, it carefully limits the expressiveness of this language, *e.g.,* by not allowing nesting of temporal operators, while allowing efficient verification of the proof obligations using SMT solvers.

Another main difference is that unlike most existing methods, Timepiece uses time as an *explicit* variable t in the language of invariants. This serves two distinct but related purposes. First, t provides a well-founded ordering to ensure that our proof rule is sound. Second, using t explicitly in the language of invariants avoids choosing some static ordering over the components, which could be otherwise used to break a circular chain of dependencies between their assumptions (*cf.* the Circ rule in [Giannakopoulou et al. 2018]). Unfortunately, it is not always possible to determine a static ordering, especially in cases where we consider multiple destinations at once symbolically.

Others have used induction over time [Misra and Chandy 1981] or over traces in specific models such as compositions of Moore/Mealy machines [Henzinger et al. 2002; McMillan 1997] and reactive modules [Alur and Henzinger 1999; Henzinger et al. 2000] to prove soundness of circular assume-guarantee proof rules. However, to the best of our knowledge, no prior efforts use time explicitly in the language of assumptions. Although handling time explicitly could be more costly for decision procedures, in practice we use abstractions (via temporal operators) that result in fairly compact formulas. Our evaluations show that these formulas can be handled well by modern SMT solvers.

There have also been many efforts that *automatically* derive assumptions for compositional reasoning [Giannakopoulou et al. 2018]. Representative techniques include computing fixed points over localized assertions called *split* invariants [Cohen and Namjoshi 2007], learning-based methods [Cobleigh et al. 2003], and counterexample-guided abstraction refinement [Bobaru et al. 2008; Elkader et al. 2018]. We can view our interfaces as split invariants, since they refer to only the local state of a component. However, at this time, we depend on the users to provide them as annotations — we plan to derive them automatically in future work.

Modular verification of distributed systems. There have been many prior efforts [Desai et al. 2018; Hawblitzel et al. 2015; Jung et al. 2015; Ma et al. 2019; Padon et al. 2016; Sergey et al. 2018; Yao et al. 2021] for modular verification of distributed systems – see a recent work [Sergey et al. 2018] for other useful pointers. In general, these efforts handle much richer program logics or computational models than the network routing algebras we target; hence the required assumptions and verification tasks are more complex, and often require interactive theorem-proving. Indeed the synchronous semantics of network routing algebras [Daggitt et al. 2018] that underlies our work is more closely related to hardware designs modelled as compositions of finite state machines (FSMs), where a component FSM's state at time $t + 1$ depends on its state at time t and new inputs at time $t + 1$, some of which could be outputs from other FSM components, *i.e.,* their state at time t. As aforementioned, no existing efforts for such models (*e.g.,* [Henzinger et al. 2002; McMillan 1997]) consider time explicitly in the assumptions.

8 CONCLUSION

Ensuring correct routing is critical to the operation of reliable networks. With the rise of cloud provider networks with hundreds of thousands of nodes, we need modular control plane verification techniques that are general, expressive and efficient. We propose Timepiece, a radical new approach for verifying network routing based on a temporal foundation, which splits the network into small modules to verify efficiently in parallel. To carry out verification, users provide Timepiece with local interfaces using temporal operators. We proved that Timepiece is sound and complete with respect to the network semantics, and argue that its temporal foundation is an excellent choice for a modular verification procedure.

REFERENCES

Anubhavnidhi Abhashkumar, Aaron Gember-Jacobson, and Aditya Akella. 2020. Tiramisu: Fast multilayer network verification. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. 201–219. https://www.usenix.org/system/files/nsdi20-paper-abhashkumar.pdf.

Anubhavnidhi Abhashkumar, Kausik Subramanian, Alexey Andreyev, Hyojeong Kim, Nanda Kishore Salem, Jingyi Yang, Petr Lapukhov, Aditya Akella, and Hongyi Zeng. 2021. Running BGP in Data Centers at Scale. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, 65–81. https://www.usenix.org/conference/nsdi21/presentation/abhashkumar https://www.usenix.org/conference/nsdi21/presentation/abhashkumar.

Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. 2008. A Scalable, Commodity Data Center Network Architecture. In *SIGCOMM*. https://doi.org/10.1145/1402946.1402967 https://dl.acm.org/doi/10.1145/1402946.1402967.

Tim Alberdingk Thijm, Ryan Beckett, Aarti Gupta, and David Walker. 2022. Kirigami, the Verifiable Art of Network Cutting. In *Proceedings of the 30th IEEE International Conference on Network Protocols (ICNP 2022)*. https://icnp22.cs.ucr.edu/assets/papers/icnp22-final111.pdf.

Rajeev Alur and Thomas A Henzinger. 1999. Reactive modules. *Formal methods in system design* 15, 1 (1999), 7–48. https://doi.org/10.1023/A:1008739929481 https://doi.org/10.1023/A:1008739929481.

Clark Barrett and Cesare Tinelli. 2018. Satisfiability modulo theories. In *Handbook of model checking*. Springer, 305–343. https://doi.org/10.1007/978-3-319-10575-8_11 https://doi.org/10.1007/978-3-319-10575-8_11.

Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. 2017a. A General Approach to Network Configuration Verification. In *SIGCOMM*. https://doi.org/10.1145/3098822.3098834 https://doi.org/10.1145/3098822.3098834.

Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. 2018. Control Plane Compression. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication* (Budapest, Hungary) *(SIGCOMM '18)*. ACM, New York, NY, USA, 476–489. https://doi.org/10.1145/3230543.3230583 https://doi.org/10.1145/3230543.3230583.

Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. 2019. Abstract Interpretation of Distributed Network Control Planes. *Proc. ACM Program. Lang.* 4, POPL, Article 42 (dec 2019), 27 pages. https://doi.org/10.1145/3371110 https://doi.org/10.1145/3371110.

Ryan Beckett and Ratul Mahajan. 2020. A General Framework for Compositional Network Modeling. In *Proceedings of the 19th ACM Workshop on Hot Topics in Networks*. 8–15. https://doi.org/10.1145/3422604.3425930.

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*. https://doi.org/10.1145/2934872.2934909.

Ryan Beckett, Ratul Mahajan, Todd Millstein, Jitu Padhye, and David Walker. 2017b. Network Configuration Synthesis with Abstract Topologies. In *PLDI*. https://doi.org/10.1145/3062341.3062367.

Mihaela Gheorghiu Bobaru, Corina S. Pasareanu, and Dimitra Giannakopoulou. 2008. Automated Assume-Guarantee Reasoning by Abstraction Refinement. In *Computer Aided Verification (CAV), Proceedings (Lecture Notes in Computer Science, Vol. 5123)*. Springer, 135–148.

CISCO. 2005. Using BGP Community Values to Control Routing Policy in Upstream Provider Network. https://www.cisco.com/c/en/us/support/docs/ip/border-gateway-protocol-bgp/28784-bgp-community.html https://www.cisco.com/c/en/us/support/docs/ip/border-gateway-protocol-bgp/28784-bgp-community.html.

Jamieson M. Cobleigh, Dimitra Giannakopoulou, and Corina S. Pasareanu. 2003. Learning Assumptions for Compositional Verification. In *Tools and Algorithms for the Construction and Analysis of Systems, TACAS, Proceedings (Lecture Notes in Computer Science, Vol. 2619)*. Springer, 331–346.

Ariel Cohen and Kedar S. Namjoshi. 2007. Local Proofs for Global Safety Properties. In *Computer Aided Verification (CAV), Proceedings (Lecture Notes in Computer Science, Vol. 4590)*. Springer, 55–67.

Matthew L Daggitt, Alexander JT 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*. ACM, 103–116. https://doi.org/10.1145/3230543.3230561 https://doi.org/10.1145/3230543.3230561.

Ankush Desai, Amar Phanishayee, Shaz Qadeer, and Sanjit A. Seshia. 2018. Compositional programming and testing of dynamic distributed systems. *PACMPL* 2, OOPSLA (2018), 159:1–159:30.

Karam Abd Elkader, Orna Grumberg, Corina S. Pasareanu, and Sharon Shoham. 2018. Automated circular assume-guarantee reasoning. *Formal Aspects of Computing* 30, 5 (2018), 571–595.

Pete Evans. 2022. Rogers says services mostly restored after daylong outage left millions offline. https://www.cbc.ca/news/business/rogers-outage-cell-mobile-wifi-1.6514373.

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*. https://www.usenix.org/system/files/conference/osdi16/osdi16-fayaz.pdf.

Nick Feamster and Hari Balakrishnan. 2005. Detecting BGP Configuration Faults with Static Analysis. In *NSDI*. https://www.usenix.org/legacy/events/nsdi05/tech/feamster/feamster.pdf.

Cormac Flanagan and Shaz Qadeer. 2003. Thread-modular model checking. In *International SPIN Workshop on Model Checking of Software*. Springer, 213–224. https://doi.org/10.1007/3-540-44829-2_14 https://doi.org/10.1007/3-540-44829-2_14.

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*. https://www.usenix.org/system/files/conference/nsdi15/nsdi15-paper-fogel.pdf.

Aaron Gember-Jacobson, Raajay Viswanathan, Aditya Akella, and Ratul Mahajan. 2016. Fast Control Plane Analysis Using an Abstract Representation. In *SIGCOMM*. https://doi.org/10.1145/2934872.2934876 https://doi.org/10.1145/2934872.2934876.

Dimitra Giannakopoulou, Kedar S Namjoshi, and Corina S Păsăreanu. 2018. Compositional reasoning. In *Handbook of Model Checking*. Springer, 345–383. https://doi.org/10.1007/978-3-319-10575-8_12 https://doi.org/10.1007/978-3-319-10575-8_12.

Nick Giannarakis, Devon Loehr, Ryan Beckett, and David Walker. 2020. NV: An Intermediate Language for Verification of Network Control Planes. In *PLDI*. Association for Computing Machinery, New York, NY, USA, 958–973. https://doi.org/10.1145/3385412.3386019 https://doi.org/10.1145/3385412.3386019.

Timothy G. Griffin, F. Bruce Shepherd, and Gordon Wilfong. 2002. The Stable Paths Problem and Interdomain Routing. *IEEE/ACM Trans. Networking* 10, 2 (2002). https://ieeexplore.ieee.org/abstract/document/993304.

Timothy G. Griffin and Joäo Luís Sobrinho. 2005. Metarouting. In *SIGCOMM*. 1–12. https://doi.org/10.1145/1080091.1080094 10.1145/1080091.1080094.

Orna Grumberg and David E Long. 1994. Model checking and modular verification. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 16, 3 (1994), 843–871. https://doi.org/10.1145/177492.177725 https://doi.org/10.1145/177492.177725.

Ashutosh Gupta, Corneliu Popeea, and Andrey Rybalchenko. 2011. Threader: A Constraint-Based Verifier for Multi-threaded Programs. In *Computer Aided Verification (CAV). Proceedings (Lecture Notes in Computer Science, Vol. 6806)*. Springer, 412–417.

Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath T. V. Setty, and Brian Zill. 2015. IronFleet: proving practical distributed systems correct. In *Proceedings of the Symposium on Operating Systems Principles, SOSP*. ACM, 1–17.

C. Hedrick. 1988. Routing Information Protocol. Internet Request for Comments. https://datatracker.ietf.org/doc/html/rfc1058 https://datatracker.ietf.org/doc/html/rfc1058.

Thomas A Henzinger, Shaz Qadeer, and Sriram K Rajamani. 1998. You assume, we guarantee: Methodology and case studies. In *International Conference on Computer Aided Verification*. Springer, 440–451. https://doi.org/10.1007/BFb0028765 https://doi.org/10.1007/BFb0028765.

Thomas A. Henzinger, Shaz Qadeer, and Sriram K. Rajamani. 2000. Decomposing Refinement Proofs Using Assume-Guarantee Reasoning. In *Proceedings of the 2000 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE Computer Society, 245–252.

Thomas A. Henzinger, Shaz Qadeer, Sriram K. Rajamani, and Serdar Tasiran. 2002. An assume-guarantee rule for checking simulation. *ACM Transactions on Programming Languages and Systems* 24, 1 (2002), 51–64.

Internet2. 2013. About Internet2. https://meetings.internet2.edu/media/medialibrary/2013/08/01/AboutInternet2.pdf.

Karthick Jayaraman, Nikolaj Bjørner, Jitu Padhye, Amar Agrawal, Ashish Bhargava, Paul-Andre C Bissonnette, Shane Foster, Andrew Helwer, Mark Kasten, Ivan Lee, Anup Namdhari, Haseeb Niaz, Aniruddha Parkhi, Hanukumar Pinnamraju, Adrian Power, Neha Milind Raje, and Parag Sharma. 2019a. Validating Datacenters at Scale. In *Proceedings of the ACM Special Interest Group on Data Communication* (Beijing, China) *(SIGCOMM '19)*. Association for Computing Machinery, New York, NY, USA, 200–213. https://doi.org/10.1145/3341302.3342094 https://doi.org/10.1145/3341302.3342094.

Karthick Jayaraman, Nikolaj Bjørner, Jitu Padhye, Amar Agrawal, Ashish Bhargava, Paul-Andre C Bissonnette, Shane Foster, Andrew Helwer, Mark Kasten, Ivan Lee, Anup Namdhari, Haseeb Niaz, Aniruddha Parkhi, Hanukumar Pinnamraju, Adrian Power, Neha Milind Raje, and Parag Sharma. 2019b. Validating Datacenters at Scale (Presentation at SIGCOMM 2019). https://conferences.sigcomm.org/sigcomm/2019/files/slides/paper_5_1.pptx.

Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL*. 637–650.

Peyman Kazemian, Michael Chang, Hongyi Zeng, George Varghese, Nick McKeown, and Scott Whyte. 2013. Real Time Network Policy Checking Using Header Space Analysis. In *NSDI*. 99–112. https://www.usenix.org/system/files/conference/nsdi13/nsdi13-final8.pdf.

Ahmed Khurshid, Xuan Zou, Wenxuan Zhou, Matthew Caesar, and P. Brighten Godfrey. 2013. VeriFlow: Verifying Network-Wide Invariants in Real Time. In *NSDI*. https://www.usenix.org/system/files/conference/nsdi13/nsdi13-final100.pdf.

R. P. Kurshan. 1988. Reducibility in analysis of coordination. In *Discrete Event Systems: Models and Applications (LNCIS, Vol. 103)*. Springer, 19–39.

K Rustan M Leino. 2010. Dafny: An automatic program verifier for functional correctness. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*. Springer, 348–370. https://doi.org/10.1007/978-3-642-17511-4_20 https://doi.org/10.1007/978-3-642-17511-4_20.

Alessio Lomuscio, Ben Strulo, Nigel Walker, and Peng Wu. 2010. Assume-guarantee reasoning with local specifications. In *International conference on formal engineering methods*. Springer, 204–219. https://doi.org/10.1007/978-3-642-16901-4_15

Nuno P. Lopes, Nikolaj Bjørner, Patrice Godefroid, Karthick Jayaraman, and George Varghese. 2015. Checking Beliefs in Dynamic Networks. In *NSDI*. https://www.usenix.org/system/files/conference/nsdi15/nsdi15-paper-lopes.pdf.

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. https://web.ist.utl.pt/nuno.lopes/pubs/fastplane-vmcai19.pdf.

K. Lougheed and Y. Rekhter. 1991. A Border Gateway Protocol 3 (BGP-3). Internet Request for Comments. https://datatracker.ietf.org/doc/html/rfc1267 https://datatracker.ietf.org/doc/html/rfc1267.

Haojun Ma, Aman Goel, Jean-Baptiste Jeannin, Manos Kapritsos, Baris Kasikci, and Karem A. Sakallah. 2019. I4: incremental inference of inductive invariants for verification of distributed protocols. In *SOSP*. ACM, 370–384.

Kenneth L. McMillan. 1997. A Compositional Rule for Hardware Design Refinement. In *Computer Aided Verification (CAV), Proceedings (Lecture Notes in Computer Science, Vol. 1254)*. Springer, 24–35.

Microsoft. 2021. Introduction to PLINQ. https://docs.microsoft.com/en-us/dotnet/standard/parallel-programming/introduction-to-plinq.

Ron Miller. 2022. As overall cloud infrastructure market growth dips to 24%, AWS reports slowdown. https://techcrunch.com/2022/10/28/as-overall-cloud-infrastructure-market-growth-dips-to-24-aws-reports-slowdown/.

Jayadev Misra and K. Mani Chandy. 1981. Proofs of Networks of Processes. *IEEE Transactions on Software Engineering* 7, 4 (1981), 417–426.

J. Moy. 1998. Open Shortest Path First Protocol Version 2. Internet Request for Comments. https://datatracker.ietf.org/doc/html/rfc2328 https://datatracker.ietf.org/doc/html/rfc2328.

D. Oran. 1990. OSI IS-IS Intra-domain Routing Protocol. Internet Request for Comments. https://datatracker.ietf.org/doc/html/rfc1142 https://datatracker.ietf.org/doc/html/rfc1142.

Susan S. Owicki and David Gries. 1976. Verifying Properties of Parallel Programs: An Axiomatic Approach. *Commun. ACM* 19, 5 (1976), 279–285.

Oded Padon, Kenneth L. McMillan, Aurojit Panda, Mooly Sagiv, and Sharon Shoham. 2016. Ivy: safety verification by interactive generalization. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI*. ACM, 614–630.

Ivan Pepelnjak. 2018. Valley-Free Routing in Data Center Fabrics. https://blog.ipspace.net/2018/09/valley-free-routing-in-data-center.html.

Amir Pnueli. 1984. In Transition From Global to Modular Temporal Reasoning about Programs. In *Logics and Models of Concurrent Systems - Conference proceedings (NATO ASI Series, Vol. 13)*. Springer, 123–144. https://doi.org/10.1007/978-3-642-82453-1_5 https://doi.org/10.1007/978-3-642-82453-1_5.

Santhosh Prabhu, Ali Kheradmand, Brighten Godfrey, and Matthew Caesar. 2017. Predicting Network Futures with Plankton. In *Proceedings of the First Asia-Pacific Workshop on Networking (APNet'17)*. 92–98. https://doi.org/10.1145/3106989.3106991 https://dl.acm.org/doi/10.1145/3106989.3106991.

Ilya Sergey, James R. Wilcox, and Zachary Tatlock. 2018. Programming and proving with distributed protocols. *Proceedings of ACM Programming Languages* 2, POPL (2018), 28:1–28:30.

SMT-LIB. 2010a. ArraysEx. https://smtlib.cs.uiowa.edu/theories-ArraysEx.shtml.

SMT-LIB. 2010b. FixedSizeBitVectors. https://smtlib.cs.uiowa.edu/theories-FixedSizeBitVectors.shtml.

SMT-LIB. 2010c. Ints. https://smtlib.cs.uiowa.edu/theories-Ints.shtml.

SMT-LIB. 2020. Unicode Strings. http://smtlib.cs.uiowa.edu/theories-UnicodeStrings.shtml.

João Luís Sobrinho. 2005. An Algebraic Theory of Dynamic Network Routing. *IEEE/ACM Trans. Netw.* 13, 5 (October 2005), 1160–1173. https://ieeexplore.ieee.org/abstract/document/1528502.

Tom Strickx and Jeremy Hartman. 2022. Cloudflare outage on June 21, 2022. https://blog.cloudflare.com/cloudflare-outage-on-june-21-2022/.

Alan Tang, Ryan Beckett, Karthick Jayaraman, Todd Millstein, and George Varghese. 2022. LIGHTYEAR: Using Modularity to Scale BGP Control Plane Verification. arXiv:2204.09635 [cs.NI] https://arxiv.org/abs/2204.09635.

Brandon Vigliarolo. 2022. After config error takes down Rogers, it promises to spend billions on reliability. https://www.theregister.com/2022/07/25/canadian_isp_rogers_outage/.

Konstantin Weitz. 2016. Getting Started With Bagpipe. http://www.konne.me/bagpipe/started.html.

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*. https://www.dougwoos.com/papers/bagpipe-netpl16.pdf.

Jianan Yao, Runzhou Tao, Ronghui Gu, Jason Nieh, Suman Jana, and Gabriel Ryan. 2021. DistAI: Data-Driven Automated Invariant Learning for Distributed Protocols. In *OSDI*. USENIX Association, 405–421.

Fangdan Ye, Da Yu, Ennan Zhai, Hongqiang Harry Liu, Bingchuan Tian, Qiaobo Ye, Chunsheng Wang, Xin Wu, Tianchen Guo, Cheng Jin, Duncheng She, Qing Ma, Biao Cheng, Hui Xu, Ming Zhang, Zhiliang Wang, and Rodrigo Fonseca. 2020. Accuracy, Scalability, Coverage: A Practical Configuration Verifier on a Global WAN. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication* (Virtual Event, USA) *(SIGCOMM '20)*. Association for Computing Machinery, New York, NY, USA, 599–614. https://doi.org/10.1145/3387514.3406217 https://doi.org/10.1145/3387514.3406217.

Peng Zhang, Aaron Gember-Jacobson, Yueshang Zuo, Yuhao Huang, Xu Liu, and Hao Li. 2022. Differential Network Analysis. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. USENIX Association, Renton, WA, 601–615. https://www.usenix.org/conference/nsdi22/presentation/zhang-peng https://www.usenix.org/conference/nsdi22/presentation/zhang-peng.

A PROOFS

We present the full proofs of Theorem 3.1, Corollary 3.2 and Theorem 3.3 below.

Theorem A.1 (Soundness). *Let A satisfy initial and inductive conditions. Then $\forall t \in \mathbb{N}, \forall v \in V, \sigma(v)(t) \in A(v)(t)$.*

Proof. Straightforward by induction over time. Let v be an arbitrary node in V.

At time 0, we have $\sigma(v)(0) = I_v$ by the definition of σ and $I_v \in A(v)(0)$ by the definition of an inductive invariant, so by substitution we have $\sigma(v)(0) \in A(v)(0)$.

For the inductive case, we must show that

$$\big(\forall v \in V,\ \sigma(v)(t) \in A(v)(t)\big) \Rightarrow \big(\forall v \in V,\ \sigma(v)(t+1) \in A(v)(t+1)\big)$$

We assume the antecedent (the inductive hypothesis). Consider the neighbors $u_1, \dots, u_k$ of v. By the definition of an inductive invariant (6), we have:

$$\forall s_1 \in A(u_1)(t), \forall s_2 \in A(u_2)(t), \dots, \forall s_n \in A(u_n)(t),$$
$$\left(I_v \oplus \bigoplus_{i \in \{1,\dots,n\}} f_{u_i v}(s_i) \right) \in A(v)(t+1)$$

Then by our inductive hypothesis, we have that $\sigma(u_i)(t) \in A(u_i)(t)$ for all u_i, so we can instantiate the universal quantifiers with these routes and substitute, which gives us:

$$I_v \oplus \bigoplus_{i \in \{1,\dots,n\}} f_{u_i v}(\boxed{\sigma(u_i)(t)}) \in A(v)(t+1)$$

The left-hand side is equal to the definition of $\sigma(v)(t+1)$ in (4), so we have what we wanted to show:

$$\boxed{\sigma(v)(t+1)} \in A(v)(t+1)$$

Then $\forall t \in \mathbb{N}, \forall v \in V, \sigma(v)(t) \in A(v)(t)$. □

Corollary A.2 (Safety). *Let A satisfy initial and inductive conditions. Let P satisfy the safety condition with respect to A. Then $\forall t \in \mathbb{N}, \forall v \in V, \sigma(v)(t) \in P(v)(t)$.*

Proof. Let v be a node and t be a time. By Theorem 3.1, $\sigma(v)(t) \in A(v)(t)$. By the safety condition (7), $A(v)(t) \subseteq P(v)(t)$. Then $\sigma(v)(t) \in P(v)(t)$, *i.e.*, P holds for v at t. □

Theorem A.3 (Completeness). *Let σ be the state of the network. Then $A(v)(t) = \{\sigma(v)(t)\}$ for all $v \in V$ and all $t \in \mathbb{N}$ satisfies the initial and inductive conditions.*

Proof. Let A be a function from nodes and times to singleton sets of routes such that $A(v)(t) = \{\sigma(v)(t)\}$ for all $v \in V$ and all $t \in \mathbb{N}$. Let v be an arbitrary node in V and let t be an arbitrary time. We want to show that $\{\sigma(v)(t)\}$ is an inductive invariant.

Starting with the initial condition case, at time 0, we have $\sigma(v)(0) = I_v$ by the definition of σ. Since $\sigma(v)(0) \in \{\sigma(v)(0)\}$, the initial condition holds.

For the inductive condition case, we want to show that:

$$\forall s_1 \in \boxed{\{\sigma(u_1)(t)\}}, \forall s_2 \in \boxed{\{\sigma(u_2)(t)\}}, \dots, \forall s_n \in \boxed{\{\sigma(u_n)(t)\}},$$
$$\left(I_v \oplus \bigoplus_{i \in \{1,\dots,n\}} f_{u_i v}(s_i) \right) \in \boxed{\{\sigma(v)(t+1)\}}$$

Checking set inclusion of $s \in \{\sigma(v)(t)\}$ is equivalent to checking that $s = \sigma(v)(t)$, so we can simplify the expression further by substituting $\sigma(u_i)(t)$ for s_i:

$$\left(I_v \oplus \bigoplus_{i \in \{1,\ldots,n\}} f_{u_i v}(\boxed{\sigma(u_i)(t)}) \right) \boxed{= \sigma(v)(t+1)}$$

This is now simply (4) flipped, so the inductive condition holds.

Then A is an inductive invariant. □