Kirigami, the Verifiable Art of Network Cutting

Timothy Alberdingk Thijm
Princeton University
Princeton, USA
tthijm@cs.princeton.edu

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

Aarti Gupta
Princeton University
Princeton, USA
aartig@cs.princeton.edu

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

***Abstract*—Satisfiability Modulo Theories (SMT)-based analysis allows exhaustive reasoning over complex distributed control plane routing behaviors, enabling verification of routing under arbitrary conditions. To improve scalability of SMT solving, we introduce a modular verification approach to network control plane verification, where we cut a network into smaller fragments. Users specify an annotated cut which describes how to generate these fragments from the monolithic network, and we verify each fragment independently, using these annotations to define assumptions and guarantees over fragments akin to assume-guarantee reasoning. We prove this modular network verification procedure is sound and complete with respect to verification over the monolithic network. We implement this procedure as Kirigami, an extension of NV [25] — a network verification language and tool — and evaluate it on industrial topologies with synthesized policies. We observe a 10x improvement in end-to-end NV verification time, with SMT solve time improving by up to 6 orders of magnitude.**

***Index Terms*—modular verification, network control plane, control plane verification, routing protocols**

I. Introduction

Today's networks are labyrinthine and hard-to-analyze systems. To determine the best paths routers may use to forward traffic, networks typically run distributed routing protocols. Despite advances like software-defined networking, these protocols remain widely used in data centers [40] and wide-area networks. Millions of lines of decentralized, low-level router configuration code control protocol behaviors, and operators must update these device configurations over time. This overwhelming complexity has led to several notable and costly outages [47], [52], [54], [55]. Often, the culprits behind these incidents are subtle network misconfigurations.

In response, researchers have developed a variety of verification tools and techniques to catch errors before outages occur. Some [7], [34], [36]–[38], [43], [46], [49] have targeted the network *data plane*, which is responsible for forwarding traffic from point A to point B. This work has produced scalable, efficient methods for modeling the data plane and checking properties of how packets traverse it.

The data plane is produced by the *control plane*. It uses the aforementioned routing protocols to decide which routes forwarding should use. Occasionally, these protocols may update their choice of routes — *e.g.,* following a device failure — and recompute new paths. When this happens, the data plane is regenerated, and the user must repeat any data plane analysis. Control plane errors can lead to further issues like route flapping, leaving human operators to hunt for subtle bugs in a Kafkaesque morass of router configurations.

To address this problem, researchers have developed another suite of tools to analyze the control plane [1], [9]–[11], [17], [19], [22], [24], [25], [50], [57], [58]. Control plane analyses consider which routes the data plane will use in given network environments, and check properties of the network in such environments. One branch of control plane verification, starting from Minesweeper [9], encodes a network as a Satisfiability Modulo Theories (SMT) formula and then asks an SMT solver [8] to check properties of the encoded network. SMT-based verification has some advantages over other approaches: it is expressive and can reason symbolically about network behavior, allowing analyses about *all possible routes* a neighbor might announce; it also may form a basis for network synthesis and repair [21]. Unfortunately, it suffers from scalability issues. Prior work has explored using abstractions to resolve this problem, *e.g.,* using symmetries in topologies to compress networks [10], [24]. These abstractions offer some relief, but cannot always handle arbitrary networks.

Control plane verification users thus face a trade-off: they may use semi-symbolic or simulation-based tools [1], [11], [22], [25], [44], [50], [58] to analyze industrial-sized networks when the flexibility of SMT-based symbolic reasoning is not necessary; or they must contend with SMT-based verifiers which may not scale to networks with more than a few hundred nodes. This paper offers another option: using a user's own insights about their network's behavior, we leverage the inherent modularity of the control plane to *cut* a monolithic network into multiple fragments to verify independently. Networks' modular structure — where end-to-end behaviors emerge from individual routers' local decisions — makes cutting an intuitive way to scale verification. In an SMT-based context, it allows us to verify properties in the presence of faults or arbitrary external announcements, which is not shown with prior abstraction approaches [10], [11]. Building on assume-guarantee verification of modular programs [23], [32], we present a new technique for modular verification of control planes and implement it as Kirigami, an extension for the NV [25] network verification tool. While we focus on SMT-based verification, one could combine our cutting technique

This work was supported in part by the National Science Foundation awards NeTS 1704336, FMitF 1837030, SHF 2107138, and Facebook Research Award on "Network control plane verification at scale."

978-1-6654-8234-9/22/$31.00 ©2022 IEEE

with other methods *e.g.,* model checking, simulation.

In a typical assume-guarantee verification approach, one can verify a safety property over a system of concurrent processes by verifying local properties of each process independently, using *assumptions* over the process's inputs and *guarantees* over its outputs. The verifier will check the required proof obligations on each component (formulated as assume-guarantee rules): if all checks pass, then the property holds for the monolithic system. Our verification technique mirrors this idea: we verify a property over network fragments (*cf.* processes), given assumptions over the rest of the network and guarantees over our fragments, to conclude that the property holds for the monolithic network.

We start from an existing model for distributed routing, the Stable Routing Problem (SRP) [10]. In an SRP, each node of the network exchanges routes with its neighbors to compute a locally-stable solution. Like other work in control plane verification [1], [11], [22], [44], we focus on networks (*i.e.,* SRPs) with unique solutions. To define an SRP, we require complete knowledge of the network and its configurations. In theory, our work could apply to interdomain routing — if multiple organizations gave us their configurations, we could jointly analyze those configurations. In practice, operators are reluctant to share their configurations outside their organization. Thus, our work's main practical application is on networks controlled by a single entity. This is frequently the case in large data centers, many of which run distributed routing protocols such as BGP [2].

We first generalize SRPs to "open SRPs", in which a network receives routes along a set of *input nodes* and sends out routes along a different set of *output nodes*. We identify the input node solutions as our open SRP's assumptions, and the output node solutions as its guarantees. We present a procedure CUT which, given an *interface* — a mapping from a cut-set of edges to routes — cuts an open SRP S into two open SRPs T_1 and T_2 covering S, and where we replace each cut edge with a route assumed in one SRP and guaranteed in the other. Interfaces can follow a network's natural boundaries, *e.g.,* tiers or hierarchies in a data center topology [3], [30], [31].

As with the traditional (closed) SRP, we can check that an open SRP satisfies a given safety property P by verifying that P holds for the SRP's solutions. We prove that if P holds on T_1 and T_2's solutions, then it holds on S's. This is the basis for our modular network verification technique. Starting from a network S, an interface I, and a safety property P, we use $\text{CUT}(S, I)$ to obtain a set of N open SRPs $T_1, \ldots, T_N$ that we verify independently. We verify P and T_i's guarantees for each open SRP T_i: if either the property or interface's guarantees do not hold, we return a counterexample demonstrating the solution that does not satisfy P or I.

SMT-based verification time can — depending on the policy and property — grow exponentially with the size of the network [9]. Hence, verifying P on each open SRP T_i takes a fraction of the time to verify P directly on S, and is embarrassingly parallel. Our experiments demonstrate that this modular verification technique works well for a variety of data center, random and backbone networks, with significant improvements in SMT solve time: we show for one set of fattree [3] benchmarks that verifying the fattree pod-by-pod cuts SMT time from 90 minutes to under 2 seconds; verifying every node individually reduces SMT time to around 10 milliseconds. Taking advantage of parallelism also cuts down NV end-to-end verification time for our largest benchmarks from over 2 hours to under 15 minutes. This modularity can scale verification to tomorrow's networks, and produce localized errors when verification fails, empowering network operators with stronger safety and reliability guarantees.

In summary, we make the following contributions:

- **A Theory of Network Fragments.** We develop an extension of the Stable Routing Problem (SRP) model [10] for network fragments. Our extension provides a method to cut monolithic SRPs into a set of fragments. We define *interfaces* to cut SRPs and map the cut edges to annotations which then define *assumptions* and *guarantees* of our fragments. We prove that under these assumptions, if these guarantees hold, then a property that holds in every fragment also holds in the monolithic network. (§IV)
- **A Modular Network Verification Technique.** We present a checking procedure to verify SRP properties. Given a property P we wish to verify, we cut an SRP S according to a given interface I into fragments, and generate checks on each fragment to both verify that our interface soundly captures the monolithic network behavior, and verify P on every fragment. This enables a novel approach for modular control plane verification based on assume-guarantee reasoning. (§V)
- **Fast, Scalable and Modular SMT Verification.** We implement our technique as Kirigami, an extension for NV, a network verification language and tool [25]. Kirigami improves on NV verification scalability and performance, with an SMT solve time up to *six orders of magnitude* faster for a selection of NV benchmarks. (§VI and §VII)

II. Overview

The Stable Routing Problem. A network is a graph with nodes V representing routers and edges E representing the links between them. A distributed control plane uses routing protocols to determine paths to routing destinations. Each router deploys its own local rules to broadcast routing announcements (or *routes*) and select a "best" route: the details of these rules vary with the protocol, but generally protocols focus on minimizing routing costs.

These elements — nodes and edges, a set of routes, and a set of rules to initialize, compare and broadcast them — form the basis for our control plane routing model, the SRP [10]. In a well-designed network, this exchange of routes eventually converges to a *stable state*, where no node may improve on its current best route by selecting another offered by a neighbor. A *solution* $\mathcal{L}$ to the SRP is a mapping from nodes to these stable routes. While routing can diverge (*i.e.,* have no solution) or converge to multiple solutions, many typical networks have

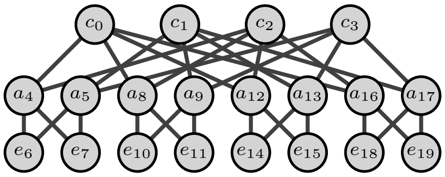


Fig. 1: A 4-pod fattree topology.

```
type attribute = { id: int; cost: int }

symbolic d : int (* symbolic id *)
require (d = 6 || d = 7 || d = 10 || d = 14 || d
      = 15 || d = 18 || d = 19)
let nodes = 20 (* topology *)
let edges = {
  0=4; 0=8; 0=12; 0=16;(*...*)
  16=18; 16=19; 17=18; 17=19;
}

let merge node x y = if x.cost < y.cost then x
    else y
let trans edge x = {x with cost = x.cost + 1}
let init node = match node with
  | 0n -> if (d = 0) then {id=d; cost=0;} else
    NULL
  | 1n -> if (d = 1) then {id=d; cost=0;} else
    NULL (*...*)
```

Fig. 2: An NV program `fat.nv` representing Fig. 1.

unique solutions (*e.g.,* when routing costs strictly increase with distance to the destination [22], [44]): we focus in this paper on such networks, like other work [1], [11], [22], [44].

An Example SRP. Consider a fattree [3] data center network, as shown in Fig. 1. Routing in fattree networks typically follows a Λ shape: traffic that starts from an edge layer switch $(e_6, \ldots, e_{19})$ travels up along a link to an aggregation layer switch $(a_4, \ldots, a_{17})$, then ascends from the pod to a core layer switch $(c_0, \ldots, c_3)$ in the spine and descends into another pod.

Suppose we wish to verify that every node in a fattree SRP instance S can reach every edge layer node of the fattree, where S is running BGP (the Border Gateway Protocol) [40]. We can do so by first modeling S's routes as highly-simplified BGP announcements $\langle p, x \rangle$ with 2 fields: an identifier p (*cf.* a prefix) and a cost metric x (abstracting, *e.g.,* local preference, AS path length [12], [51], *etc.*).[1] Each node has an identifier p, where every node has an initial route $\langle p, 0 \rangle$ for its identifier, and no route to other identifiers. Nodes will broadcast their current routes to their neighbors: in this simple example, if a node has a route $\langle p, x \rangle$, it will send a route $\langle p, x + 1 \rangle$ to its neighbors (incrementing the metric). Nodes compare each received route with their current choice and select the one with the smallest cost, and then re-broadcast if their route changes. A node u's solution $\mathcal{L}(u)$ is the best route between u's initial route and the solutions broadcast by each of u's neighbors.

Verifying SRPs with NV [25]. To verify all-edge reachability in S, we must check that for any choice of identifier p of an edge layer node, all nodes of the network have a path of cost at most 4 to that node. Naively enumerating all possible identifiers is obviously inefficient, if not infeasible in practice. In some scenarios, an equivalence class-based approach like that of Plankton [50] may make the space of all identifiers small enough to efficiently enumerate. We will use a symbolic approach, where we treat the identifier as a symbolic variable d: we then will verify that for any concrete identifier instantiating d, every node can reach that identifier. Symbolic variables can also help verify properties in the presence of link failures or arbitrary external announcements from outside one's network. One verification tool supporting symbolic reasoning is NV [25]. NV is a functional programming language for modeling control planes and verifying their properties using SMT. An NV program's components resemble an SRP's: it has a topology with `nodes` and `edges`; a type of routes `attribute`; a function `init` to initialize routes; a function `trans` to broadcast routes; and a function `merge` to compare routes. NV provides `symbolic` and `require` expressions to declare and constrain symbolic values, respectively. Fig. 2 presents a condensed NV program for Fig. 1.

```
1 include "fat.nv"
2
3 (* map nodes to solutions (stable routes) *)
4 let sol = solution {init = init; trans = trans;
      merge = merge}
5 (* check a property of every node's solution *)
6 assert foldNodes (fun n r acc -> acc && r.id = d
       && r.cost <= 4) sol true
```

Fig. 3: An NV program asserting that every node can reach every prefix advertised by an edge layer switch.

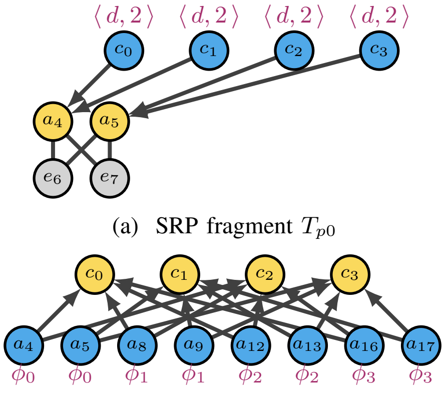
Fig. 3 demonstrates how to verify a safety property P in NV, where P holds *iff* $\forall u.\ \mathcal{L}(u).p = d \land \mathcal{L}(u).x \leq 4$. We define a solution (line 4) using `init`, `trans` and `merge` from Fig. 2. We then assert (line 6) that P holds on this solution. When we ask NV to verify Fig. 3, it encodes S and P as an SMT query, and confirms that P holds.

Scaling Up SRP Verification. SMT-based verification is flexible, but has issues when it comes to scalability. Our evaluation in §VII shows it scales superlinearly for larger fattrees with more complex policies: from 0.03 seconds for a 20-node network, to 1.4 seconds for an 80-node network, and 1833.7 seconds for a 320-node network! To verify industrial fattree networks with 10^4 or more switches [34], we need a way to scale this technique up.

Suppose then that we took a large network and *cut it into fragments*, then verified a safety property P on each fragment independently. If P holds for every fragment, then we want it to hold for the monolithic network; otherwise, we want to observe real counterexamples as in the monolithic network. To achieve this goal, our cutting procedure must also summarize the network behavior external to each fragment.

We incorporate these summaries into the SRP model by generalizing it to open SRPs. Open SRPs extend the SRP model by designating some nodes as *input nodes* and some

[1] In a real network, routes could represent many more BGP fields, but this example provides the necessary detail to demonstrate the basics of SRPs.



(a) SRP fragment T_{p0}

(b) SRP fragment T_{spines}

Fig. 4: SRP fragments T_{p0} and T_{spines}, with input nodes in blue, output nodes in yellow and assumptions in purple.

others as *output nodes*. We annotate input and output nodes with routes representing solutions *assumed* on the inputs and *guaranteed* on the outputs. We express these annotations using an *interface*: a mapping from each cut edge to a route annotation. Given an open SRP S and an interface I, we cut S into open SRP fragments, where each fragment identifies assumptions on its inputs and guarantees on its outputs.

Cutting Down Fattrees. We will now move on to demonstrating this idea for our example. Let's cut each pod of our network into its own fragment T_{p0} through T_{p3}, leaving the spine nodes as a fifth fragment T_{spines}.

Figs. 4a and 4b show pod 0 and the spines of Fig. 1 as open SRPs T_{p0} and T_{spines}, respectively. In T_{p0}, we assume routes from the spines and check guarantees on a_4 and a_5. Every route guaranteed by one fragment is assumed by another (and vice-versa): if we guarantee that c_0 has a route $\langle d, 2 \rangle$ in T_{spines}, we assume it has a route $\langle d, 2 \rangle$ in T_{p0}. The exact route advertised by an aggregation node a depends on if the destination identifier d lies in a's pod or not. For instance, if $d = 6$, nodes a_4, a_5 of pod 0 have a route $\langle d, 1 \rangle$ from their neighbor e_6, while all other aggregation nodes have a route $\langle d, 3 \rangle$ (via the core nodes). We write ϕ_i as a shorthand for this reasoning over costs in Fig. 4b, where

$$\phi_i = \text{if } d \text{ in pod } i \text{ then } \langle d, 1 \rangle \text{ else } \langle d, 3 \rangle$$

Verifying Network Fragments. In modular verification, we perform an independent verification query for each fragment: we encode the open SRP and property, along with an assumptions formula assuming a state of the inputs and a guarantees formula to check on the state of the outputs. We then submit every query to our solver and ask if the network has a solution where, under the given assumptions, either the property is false or the guarantee do not hold. The solver searches for a counterexample demonstrating a concrete violation of the property or our guarantees. Guarantee violations demonstrate possible bugs in our network implementation or mistakes in our beliefs, just as property violations do.

Let us consider our fattree network again. Suppose we misconfigured a_4 to black hole (silently drop) outgoing traffic.

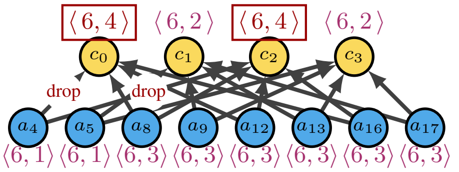
```
include "fat.nv"

let partition node = match node with
  | 0n | 1n | 2n | 3n -> 0 (* spines *)
  | 4n | 5n | 6n | 7n -> 1 (* p0 *)(*...*)

let interface edge x = match edge with
  | 0~_ | 1~_ | 2~_ | 3~_ -> x = { id = d; cost
    = 2; }
  | 4~_ | 5~_ -> x = { id = d; cost = if d > 3
    && d < 8 then 1 else 3; }
  | 8~_ | 9~_ -> x = { id = d; cost = if d > 7
    && d < 12 then 1 else 3; }
```

Fig. 5: An (abbreviated) NV program to cut Fig. 2 into pods.

Consider what happens when $d = 6$, meaning the destination is e_6: then d is in a_4's pod. Because a_4 is dropping outgoing traffic, the best route c_0 will receive will be a $\langle 6, 3 \rangle$ route from one of a_8, a_{12}, a_{16}, and hence $\mathcal{L}(c_0) = \langle 6, 4 \rangle$. The solution in T_{spines} will be as follows:



Our interface maps c_0a_8 to $\langle d, 2 \rangle$, so we must guarantee that $\mathcal{L}(c_0) = \langle d, 2 \rangle$ when verifying T_{spines}. Due to the bug, this check fails ($\langle 6, 2 \rangle \neq \langle 6, 4 \rangle$) and our solver returns the solution above as a counterexample. This localizes the counterexample to this part of the network; our other fragments will pass verification. This means that, so long as our interface remains the same, we only need to correct the bug and re-verify T_{spines}, without needing to re-verify *any other fragment*. In §IV, we prove that our cutting technique is sound with respect to verification of P on the monolithic network S: if our guarantees and P hold for all fragments of S, then P holds for the monolithic network S.

Verifiable Network Cutting with Kirigami. As part of our work, we implemented an extension Kirigami to NV for cutting and verifying networks. Fig. 5 shows an NV file with new `partition` and `interface` functions. `partition` maps each node to a fragment, while `interface` adds assertions to check that a route x along a cross-fragment edge equals the given annotation, *e.g.,* that the route from `0n` to `4n` has id d and cost 2. These functions provide the necessary detail to construct our fragments and modularly verify them.

A Cut Above the Rest. Pod-based cuts suit our hierarchical view of fattrees, but we can consider alternative cuts. We could cut Fig. 2 so that every node is in its own fragment. Verifying a single node in SMT can take milliseconds, and hence leads to significant performance improvements. The corresponding NV program resembles Fig. 5, except every node maps to its own fragment and we annotate every edge.

III. Background on the Stable Routing Problem

We summarize prior work [10] on the Stable Routing Problem (SRP) network model. Its components resemble routing algebras used for reasoning about convergence of routing protocols [13], [28], [53], but SRPs also include a network topology for reasoning about properties such as reachability between nodes.

An SRP instance S is a 6-tuple $(V, E, R, \mathsf{init}, \oplus, \mathsf{trans})$, defined as follows.

Topology. V is a set of nodes and $E \subseteq V \times V$ is a set of directed edges. We write uv for an edge from node u to node v. Edges may not be self-loops: $\forall v \in V.\ vv \notin E$.

Routes. R is a set of routes that describe the fields of routing messages. For example, when modeling BGP, R might be a set of tuples of an integer local preference, a set of community tags, and a sequence of AS numbers representing the AS path [12], [51].

Node Initialization. The initialization function $\mathsf{init} : V \to R$ describes the initial route of each node. When modeling single destination routing, init may map a destination node e_{19} to some initial route r_d, and all other nodes to a null route; in multiple destination routing, we may have many initial routes.

Route Update. The merge function $\oplus : R \times R \to R$ defines how to compare routes. $\oplus$ represents updates of a node's selected route: we assume $\oplus$ is associative and commutative, *i.e.,* the order in which routes are merged does not matter.

Route Transfer. The transfer function $\mathsf{trans} : E \times R \to R$ describes how routes are modified between nodes. Given an edge uv and a route r from node u, $\mathsf{trans}(uv, r)$ determines the route received at v.

Solutions. A solution $\mathcal{L} : V \to R$ is a mapping from nodes to routes. Intuitively, a solution is defined such that each node is *locally stable*, *i.e.,* it has no incentive to deviate from its currently chosen neighbors. Nodes compute their solution via message exchange, where each node in the SRP advertises its chosen route to each of its neighbors. Formally, an SRP solution $\mathcal{L}$ satisfies the constraint:

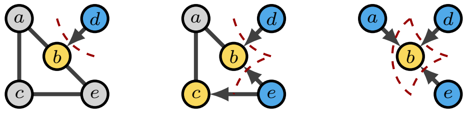
$$\mathcal{L}(v) = \mathsf{init}(v) \oplus \bigoplus_{uv \in E} \mathsf{trans}(uv, \mathcal{L}(u)) \tag{1}$$

where $\bigoplus$ is the sequence of $\oplus$ operations on each transferred route $\mathsf{trans}(uv, \mathcal{L}(u))$ from each neighbor u of v. These received routes are merged with v's initial value $\mathsf{init}(v)$.

A solution may determine an SRP's forwarding behavior or another decision-making procedure, as shown in [10]. We omit discussing forwarding behavior to focus on a general SRP definition without restricting ourselves only to forwarding.

IV. Cutting SRPs

We now introduce our original contributions, starting with *open SRPs*. We define a Cut procedure to partition an open SRP into fragments. We prove soundness and completeness of fragment solutions with respect to the larger SRP's solution.



(a) SRP S (b) SRP T (c) SRP U

Fig. 6: A series of successive cuts which produce open SRPs S, T and U. Base nodes are grey, input nodes blue and output nodes yellow. Each cut (in red) slices off part of the SRP, leaving input nodes to represent the cut components.

Notation. We introduce some notation in this section. $\mathsf{dom}(f)$ is the *domain* of the function f, and $f|_X$ is the *restriction* of f to $X \subseteq \mathsf{dom}(f)$. We use subscripts to specify SRP components, *e.g.,* init_S refers to SRP S's init component.

Open SRPs. An *open SRP* generalizes our earlier SRP definition to include *assumptions* and *guarantees*. An open SRP instance S is an 8-tuple $(V, E, R, \mathsf{init}, \oplus, \mathsf{trans}, \mathsf{ass}, \mathsf{guar})$.

The first six elements are exactly as for regular (closed) SRPs. The final two elements, ass ("assumptions") and guar ("guarantees"), are *partial functions* ($V \hookrightarrow R$) mapping mutually disjoints subsets $V^{in}, V^{out} \subseteq V$ to routes. We use V^{in} (input nodes) as a shorthand for $\mathsf{dom}(\mathsf{ass})$ and V^{out} (output nodes) as a shorthand for $\mathsf{dom}(\mathsf{guar})$. All nodes that are neither input nor output nodes are "*base nodes*" V^{base}. A closed SRP is an open SRP where $V^{in} = V^{out} = \varnothing$. Going forward, we assume an open SRP wherever we write "SRP".

Input nodes are *source nodes* (*i.e.,* they have in-degree 0). Hence, they act as auxiliary nodes, indicating where a fixed incoming route "arrives" from *outside* the SRP, as specified by the assumptions ass. Output nodes correspondingly mark where routes "depart" the SRP, per the guarantees guar. We do not require any connectivity properties of output nodes: they simply identify an outgoing route we wish to guarantee, without detailing where the SRP is announcing that route to.[2] Fig. 6 illustrates this concept.

Definition IV.1 (Open SRPs)**.** *An* open SRP instance $S = (V, E, R, \mathsf{init}, \oplus, \mathsf{trans}, \mathsf{ass}, \mathsf{guar})$ *has the following properties:*

- $V = V^{in} \cup V^{out} \cup V^{base}$ *and* $V^{in}, V^{out}, V^{base}$ *are pairwise-disjoint;*
- $\mathsf{ass} : V^{in} \to R$ *and* $\mathsf{guar} : V^{out} \to R$*; and*
- $\forall v \in V^{in}.\ \textit{in-degree}(v) = 0.$

Open SRP Solutions. A mapping $\mathcal{L} : V \to R$ is a *solution* to an open SRP *iff*:

$$\mathcal{L}(u) = \mathsf{init}(u) \oplus \bigoplus_{vu \in E} \mathsf{trans}(vu, \mathcal{L}(v)) \qquad \forall v \notin V^{in} \tag{2}$$

$$\mathcal{L}(u) = \mathsf{ass}(u) \qquad \forall v \in V^{in} \tag{3}$$

$$\mathcal{L}(u) = \mathsf{guar}(u) \qquad \forall v \in V^{out} \tag{4}$$

Note that Equations (2) and (4) both apply for all outputs $v \in V^{out}$. Solutions for open SRPs resemble closed SRP solutions, with the addition of constraints based on the values of ass and

[2] We could have alternatively attached auxiliary nodes to output nodes to show where routes went, but we found this definition more intuitive.

guar. For any input node u, its assumption $\mathsf{ass}(u)$ determines the node's solution directly; for an output node u, its solution $\mathcal{L}(u)$ must be consistent with both the right-hand side of (2) *and* the right-hand side of (4). Hence, if $\exists u \in V^{out}.\ \mathsf{init}(u) \oplus \bigoplus_{vu \in E} \mathsf{trans}(vu, \mathcal{L}(v)) \neq \mathsf{guar}(u)$, there is no solution to the open SRP. We focus on open SRPs with unique solutions.

We consider *safety properties* on SRP solutions. A property $P : V \to 2^R$ holds on an SRP S *iff*

$$\forall v \in V_S^{base} \cup V_S^{out}.\ \mathcal{L}(v) \in P(v) \tag{5}$$

We ignore input nodes as they are "outside" the SRP. We can express many properties this way, including reachability, isolation, path length, waypointing and fault tolerance [9], but not convergence properties or properties over multiple nodes, *e.g.,* that two nodes u, v have the same solutions $\mathcal{L}(u) = \mathcal{L}(v)$ (useful for checking consistency across nodes, irrespective of their exact solutions).

Interfaces and Cutting SRPs. We now consider how to cut an SRP S into two SRPs T_1 and T_2, where T_1 and T_2 cover S and replicate its behavior using their assumptions and guarantees. We select a cut-set $C \subseteq E$ of edges in S and annotate each cut edge uv with a route that describes the solution transferred from u to v. This cut-set divides the *non-input nodes* $V_S \setminus V_S^{in}$ into two disjoint subsets, W_1 and W_2 (we will treat input nodes separately). We call this annotated cut-set an *interface* I.

Definition IV.2 (Interface)**.** *Let S be an SRP and let $C \subseteq E$ be a cut-set partitioning $V_S \setminus V_S^{in}$. $I : C \to R_S$ is an* interface *if it maps every element uv of C to a route $I(uv)$ in R_S.*

We now define a CUT procedure. Given an SRP S and an interface I, $\textsc{Cut}(S, I)$ partitions S into two SRPs, T_1 and T_2, which we call *fragments*. We can CUT an SRP into arbitrarily many SRPs by recursively cutting the resulting fragments.

Definition IV.3 (CUT)**.** *Let S be an SRP and let I be an interface over S. Let (W_1, W_2) be disjoint subsets of $V_S \setminus V_S^{in}$ as cut by* $\mathsf{dom}(I)$. *Given S and I, $\textsc{Cut}(S, I) = (T_1, T_2)$, where T_1 and T_2 are open SRPs where, for $i \in \{1, 2\}$:*

$$\begin{aligned}
V_i^{in} &= \{u \mid u \in V_S^{in} \land \exists uv \in E_S.\ v \in W_i\} \\
&\quad \cup \{u \mid \exists uv \in \mathsf{dom}(I).\ v \in W_i\} \\
V_i^{out} &= \{u \mid u \in W_i \land u \in V_S^{out}\} \\
&\quad \cup \{u \mid \exists uv \in \mathsf{dom}(I).\ u \in W_i\} \\
V_i &= W_i \cup V_i^{in} \\
E_i &= \{uv \mid u, v \in V_i \land uv \in E_S\} \\
R_i &= R_S \\
\oplus_i &= \oplus_S \\
\mathsf{init}_i &= \mathsf{init}_S|_{V_i} \\
\mathsf{trans}_i &= \mathsf{trans}_S|_{E_i} \\
\mathsf{ass}_i(u) &= \begin{cases} \mathsf{ass}_S(u) & \textit{if } u \in V_S^{in} \\ I(uv) & \textit{if } uv \in \mathsf{dom}(I) \land v \in V_i \end{cases} \\
\mathsf{guar}_i(u) &= \begin{cases} \mathsf{guar}_S(u) & \textit{if } u \in (V_S^{out} \setminus V_i^{in}) \\ I(uv) & \textit{if } uv \in \mathsf{dom}(I) \land v \notin V_i \end{cases}
\end{aligned}$$

The resulting SRPs T_1 and T_2 have the following properties:

- **Covering:** $V_1 \cup V_2 = V_S$ and $E_1 \cup E_2 = E_S$, with $V_1^{in} \cup V_2^{in} \supseteq V_S^{in}$ and $V_1^{out} \cup V_2^{out} \supseteq V_S^{out}$;
- **Policy preservation:** init, trans and $\oplus$ produce the same routes as in S for all possible routes in R_S;
- **Input-output nodes:** every input node u in T_1 or T_2 that is *not* an input node "inherited" from S has a corresponding output node in the other fragment, and such that $\mathsf{ass}_1(u) = \mathsf{guar}_2(u)$ or $\mathsf{ass}_2(u) = \mathsf{guar}_1(u)$;
- **Input-output nodes produced by I:** $\forall uv \in \mathsf{dom}(I)$, u is an input-output node with the above property;
- **Shared inherited inputs:** the only other nodes shared by T_1 and T_2 are input nodes into *both* fragments inherited from S: $V_1 \cap V_2 = (V_1^{in} \cap V_2^{in}) \cup (V_1^{in} \cup V_2^{in}) \setminus V_S^{in}$.

Importantly, CUT defines T_1 and T_2 to have *equal* assumptions and guarantees along each cut edge using our interface I. For each edge $uv \in \mathsf{dom}(I)$, $\textsc{Cut}(S, I)$ establishes a guarantee $\mathsf{guar}(u) = I(uv)$ in T_1 and an assumption $\mathsf{ass}(u) = I(uv)$ in T_2 (or vice-versa). By requiring guarantees and assumptions to be equal, we rely on the stability of an open SRP's solution to rule out circular (self-justifying) assumptions. As u's solution is both assumed in one fragment and guaranteed in the other, we refer to it as an *input-output node*.

We prove that if we use CUT to produce fragments T_1 and T_2 from S, then the joined solutions of T_1 and T_2 are a solution of S (soundness); and that if S has a solution, then there always exists an interface I that given to CUT produces two fragments T_1 and T_2 such that the solution of S is a solution (when appropriately restricted) for T_1 and T_2 (completeness).

Correctness. We now present theorems relating an SRP's solution to the solutions of its CUT-produced fragments. By showing that the fragments' solutions are the same as the monolithic SRP's, we can use the fragments *in place of* the monolithic SRP during verification of a property P. Proofs are available in an extended version of the paper [4].

We first prove that the solutions of the fragments T_1, T_2 are a solution to the monolithic SRP S: each node of S maps to its fragment solution, with S's input nodes mapping to their expected assumptions.

Theorem IV.1 (CUT is Sound)**.** *Let S be an open SRP, and let I be an interface over S. Let $\textsc{Cut}(S, I) = (T_1, T_2)$. Suppose T_1 has a unique solution $\mathcal{L}_1$ and T_2 has a unique solution $\mathcal{L}_2$. Consider a mapping $\mathcal{L}_S{}' : V_S \to R$, defined such that:*

$$\begin{aligned}
\forall v \in V_1.&\ \mathcal{L}_S{}'(v) = \mathcal{L}_1(v) \\
\forall v \in V_2.&\ \mathcal{L}_S{}'(v) = \mathcal{L}_2(v) \\
\forall v \in V_S^{in}.&\ \mathcal{L}_S{}'(v) = \mathsf{ass}_S(v)
\end{aligned}$$

Then $\mathcal{L}_S{}'$ is a solution of S.

We can always find a suitable interface I to cut S, such that T_1 and T_2 have the same solution as S for each node: we simply annotate each cut edge uv with the solution $\mathcal{L}_S(u)$, which is the solution transferred from u to v in S.

Algorithm 1 The fragment checking algorithm.

```
 1: proc Solve(fragment $T$, property $P$)
 2:     $N \leftarrow$ Encode($T$)                                              ▷ (2)
 3:     $A \leftarrow \bigwedge_{u \in V_T^{in}} \mathcal{L}_T(u) = \mathsf{ass}_T(u)$                      ▷ (3)
 4:     $G \leftarrow \bigwedge_{u \in V_T^{out}} \mathcal{L}_T(u) = \mathsf{guar}_T(u)$                    ▷ (4)
 5:     $Q \leftarrow \bigwedge_{u \in V_T^{base} \cup V_T^{out}} \mathcal{L}_T(u) \in P(u)$   ▷ property check
 6:     return AskSat($A \wedge N \wedge \neg(G \wedge Q)$)

 7: proc Check(SRP $S$, property $P$, interface $I$)
 8:     $T_1, \dots, T_N \leftarrow$ Cut($S, I$)
 9:     for $i \leftarrow 1, N$ do in parallel
10:         $r \leftarrow$Solve($T_i, P$)
11:         if $r \neq$ Unsat then
12:             return $r$
13:     return Unsat
```

Theorem IV.2 (Cut is Complete)**.** *Let S be an open SRP, and let I be an interface over S. Let* Cut$(S, I) = (T_1, T_2)$. *Assume S has a unique solution $\mathcal{L}_S$. Assume that $\forall uv \in \mathsf{dom}(I).\ I(uv) = \mathcal{L}_S(u)$. Consider the following two mappings ${\mathcal{L}_1}' : V_1 \to R$ and ${\mathcal{L}_2}' : V_2 \to R$, defined such that:*

$$\forall v \in V_1.\ {\mathcal{L}_1}'(v) = \mathcal{L}_S(v)$$
$$\forall v \in V_2.\ {\mathcal{L}_2}'(v) = \mathcal{L}_S(v)$$

Then ${\mathcal{L}_1}'$ is a solution for T_1 and ${\mathcal{L}_2}'$ is a solution for T_2.

Our proof of soundness implies that any property that holds over our fragments will hold over the monolithic network.

Corollary IV.3 (Cut Preserves Properties)**.** *Let S be an open SRP, and let I be an interface over S. Let* Cut$(S, I) = (T_1, T_2)$. *Let P be a safety property. Assume S has a unique solution $\mathcal{L}_S$, and that T_1 has a solution $\mathcal{L}_1$ and T_2 has a solution $\mathcal{L}_S$. Then if P holds on T_1 and P holds on T_2, P holds on S.*

V. Checking Fragments in SMT

We now present our three-step modular verification methodology: *(i)* given an SRP S and an interface I, produce N fragments using Cut(S, I); then *(ii)* in parallel, encode each fragment to SMT and check its guarantees and a safety property P under the given assumptions; and *(iii)* if any guarantees fail, let the user *refine* I or correct network bugs. If the SMT solver verifies P and all guarantees over S's fragments, we can conclude that it has verified P over S.

The Fragment Checking Algorithm. Algorithm 1 shows how we cut an SRP and check the three constraints on open SRP solutions (described in §IV) on each of the fragments. We start in the Check procedure on line 1.7. Check calls Cut(S, I) to cut S into fragments, and then calls Solve (line 1.1) on each fragment, reporting any Sat result it receives back from the solver. Solve encodes (2) on line 1.2, (3) on line 1.3, (4) on line 1.4, and the check that P holds on line 1.5. Since we want to know if G or P are ever violated, our query formula conjoins Encode(T) and A with the negation of $G \wedge Q$ (line 1.6). AskSat asks an SMT solver if this formula is satisfiable, and returns either Sat with a model, or Unsat. This model will be a mapping L from V_T to R_T where the Encode(T) and A constraints hold, but $\exists u \in V_T^{out}.\ L_T(u) \neq \mathsf{guar}_T(u)$ (guarantee violation) or $\exists u \in V_T.\ L_T(u) \notin P(u)$ (property violation). Otherwise, if the solver returns Unsat, then the guarantees and property always hold.[3]

Refining Interfaces. If every fragment returns Unsat, by Corollary IV.3, we conclude that P and G hold and the interface is *correct*. However, if any fragment returns Sat, we must determine why our property or guarantees were violated. For example, in §II, we considered if our interface correctly captured the intended network behaviour, but a bug in the network policy led to a guarantee violation. If the reverse were true — our network is configured correctly, but our interface is incorrect — we must *refine* our interface to correct it. Both cases may be common in practice, and point to the importance of *checking* our interfaces: counterexamples provide users with insight into why the network's actual behavior does not conform to their beliefs. Other annotation checking tools like Dafny [41] may use a similar interactive process of refining interfaces as the user identifies inconsistencies or bugs.

By Theorem IV.1, we know that any incorrect interface will not define a solution in T_1 and T_2, meaning our guarantee constraint in Solve fails and a counterexample is returned. This counterexample may then inform a new interface we can provide in a successive run of Check. Returning to our fattree fragments in Fig. 4, suppose we used the same interface except for an incorrect annotation $I(c_0 a_4) = \langle\, d, 1 \,\rangle$. To check the corresponding guarantee, we generate a constraint $\mathcal{L}_{spines}(c_0) = \langle\, d, 1 \,\rangle$. Solve$(T_{spines}, P)$ returns Sat, providing $\mathcal{L}_{spines}(c_0) = \langle\, d, 2 \,\rangle$ as a counterexample. We can then inspect T_{spines} to see that $\langle\, d, 1 \,\rangle$ is not a possible route given the assumptions on c_0's input nodes, and correct our interface to specify $\langle\, d, 2 \,\rangle$ instead.

VI. Implementation

We built Kirigami on top of the NV language [25]. NV represents routing protocols, such as BGP, OSPF and RIP, using an SRP-like model and a toolbox of types and data structures such as booleans, integers, tuples, records, maps and non-recursive functions. Our Kirigami extension adds `partition` and `interface` functions to NV: when we run NV on a file that declares these functions, NV cuts the SRP into fragments as described by Definition IV.3 of Cut. The `partition` function maps each node to a fragment, while the `interface` function defines the interface I.

Kirigami's SMT encoding follows Algorithm 1, using NV's monolithic SRP encoding as the encoding function Encode. Our implementation uses the OCaml Parmap library [14] to parallelize fragment-specific work, including decomposing properties and the embarrassingly-parallel Solve procedure. The only limit on parallelism is the number of CPU cores.

[3] Given a network with multiple solutions, Solve checks if any solution consistent with the assumptions from the interface also satisfies the guarantees and property; other interfaces may be necessary to cover all solutions.

VII. Evaluation

We evaluated Kirigami on a variety of NV benchmarks representing fattree, random and Internet topologies. Our questions focus on the scalability and performance of Kirigami in comparison to NV, specifically: *(i)* does Kirigami improve on NV verification time across topologies and properties, and *(ii)* how do different cuts impact Kirigami performance? We consider two metrics for verification time: the maximum time reported to verify an SMT query encoding the monolithic network or fragment using the Z3 [15] SMT solver;[4] and the "total time" of NV, which is the time taken by NV's pipeline of network transformations, partitioning (for cut networks), encoding to SMT and solving the query or queries.

We ran each benchmark on a computing cluster node with a 2.4GHz CPU and up to 128GB of memory per CPU core. For cut benchmarks, we parallelized partitioning and solving over up to 32 cores.[5] Each benchmark tested verification of either the monolithic network or a cut network. We timed out any benchmark that did not finish solving a Z3 query in 2 hours.[6]

Fattrees. To evaluate Kirigami's performance for fattrees, we made use of the shortest path policy SP and valley-free policy FAT described in [25], along with two extensions: an all-edge reachability policy AP and an original fault-tolerance policy MAINT. Whereas SP checks reachability of a fixed prefix of an edge layer node, AP verifies that all prefixes of edge layer nodes are reachable using a symbolic variable for the set of possible prefixes. MAINT extends SP by requiring that nodes avoid routing through a non-destination node `down` which is currently down for maintenance. We check this property for any down node by encoding `down` as a symbolic value. The set of routes R modelled routing using a combination of eBGP, connected and static routes: for eBGP, we represented its fields with bitvectors representing local preference, AS path length, the multi-exit discriminator (MED), and a set of integer BGP community tags; for connected and static routes, we use integers representing the next hop; we model the choice between eBGP, connected and static routing using 2 bits.

As in [25], we parameterize fattree designs by k, the number of pods: we vary the topology size from $k = 4$ (20 nodes) to $k = 20$ (500 nodes) to assess scalability.[7] Furthermore, we consider four different cuts of our fattree networks:

- **Vertical**: creates 2 fragments, each with half the spines and half the pods ($\frac{5k^2}{8}$ nodes);
- **Horizontal**: creates 3 fragments: the pod containing the routing destination (k nodes), the spines ($\frac{k^2}{4}$ nodes), and all the other pods ($k^2 - k$ nodes);
- **Pods**: creates $k + 1$ fragments (given k pods): the spine nodes, and each pod (k nodes) in its own fragment; and
- **Full**: creates $|V|$ fragments (given $|V|$ nodes), with every node in its own fragment.

To simplify the process of coming up with interfaces for evaluating these parametric networks, we wrote a script to generate interfaces for each of our policies. We computed BGP AS path lengths for each node using graph algorithms, and then generated the other route fields according to the policy's behavior.[8] For SP, we only computed shortest paths. For AP, we determined path lengths based on a node's tier relative to the symbolic destination node, as presented in §II. For FAT, we replicated how the policy sets community tags according to each pod's tier to block down-up-down routes (valleys) [20], [48]. For MAINT, we used Yen's 2-shortest paths algorithm [59]: this gives the shortest and second-shortest path (taken if `down` lies on the shortest path) to the destination from each node. We then assigned routes conditioned on the length of the shortest path avoiding `down`.

We compare SMT verification time for monolithic benchmarks versus their cut counterparts in Fig. 7. We plot the number of nodes in the monolithic benchmark against the maximum time spent by Z3 solving the SMT queries. Time is shown on a logarithmic scale. All policies show extreme improvements in SMT time as the number of fragments grows. The maximum SMT time for a full cut fragment of our largest SP benchmark is *six orders of magnitude* faster than the monolithic time. The FAT policy's SMT encoding is most complex, leading to timeouts for the monolithic FAT16 and FAT20 benchmarks: monolithic SMT solving also times out for AP20 and MAINT20. Most fragments are proportionally sized in terms of non-input nodes[9], but fragments with more input nodes such as spine fragments or with more complex policies tend to take longer to solve.

Fig. 8 plots the times for end-to-end NV verification, again on a logarithmic scale. These times include partitioning the network and encoding queries to SMT. SMT encoding and solving dominate all other operations in all benchmarks, except for fully-cut benchmarks, where partitioning is the longest-running operation (between 25–65% of NV time). Partitioning takes up to 25% of NV time for the pods and horizontal cuts: thanks to the reduced SMT time for these benchmarks, they see the best speedup relative to the monolithic benchmarks, *e.g.,* for $k = 16$ benchmarks, we see the pods cut finish 5–25 times faster than the monolithic benchmark. The pods and full cut benchmarks for FAT20 hit our memory limit with 128 GB per core. We increased available memory while decreasing cores (16 cores, 256 GB for pods; 8 cores, 512 GB for full) to handle them (results in Fig. 8c). This does not affect Z3 times, but NV times could improve with more available memory.

Random Networks. We next assess Kirigami with

[4] As we can solve each fragment SMT query independently, the maximum query time is an upper bound on the total SMT solve time when we solve every fragment in parallel.

[5] Benchmarks with $i < 32$ fragments ran in parallel on i cores. As each core could use up to 128 GB of memory, the total memory available was $128i$ GB (up to 4 TB).

[6] Cut benchmarks call Z3 multiple times, and hence had a two-hour limit on individual calls: total NV time could then exceed 2 hours.

[7] A k-fattree has $\frac{5}{4}k^2$ nodes and k^3 edges.

[8] We used Kirigami's counterexamples to debug mistakes in our reasoning when writing our script, as described in §V.

[9] The horizontal benchmark is a notable exception, where the "non-destination pods" fragment is significantly larger.

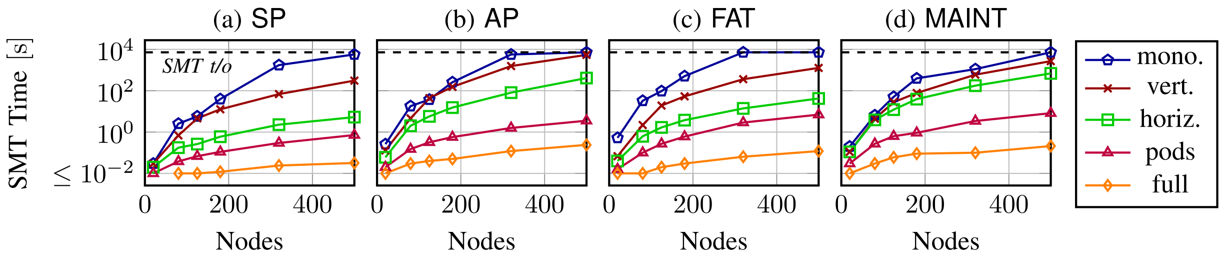


Fig. 7: Largest SMT solve times for fattree benchmarks.

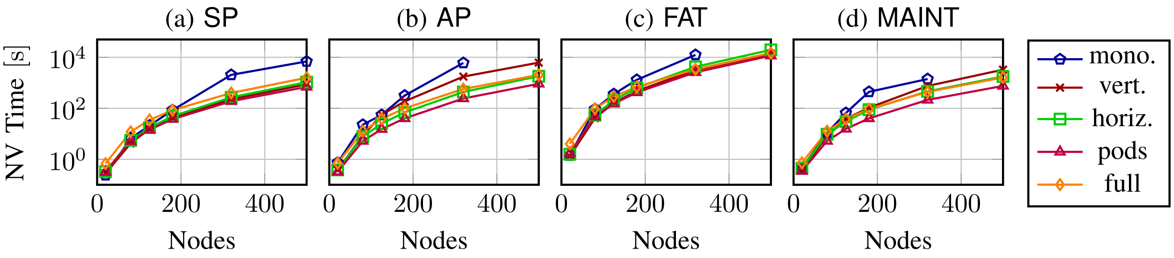


Fig. 8: NV times for fattree benchmarks.

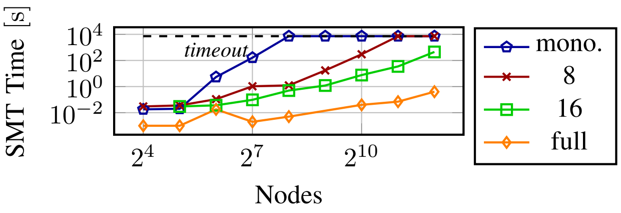


Fig. 9: Largest SMT solve times for random networks.

randomly-generated topologies of N nodes using the Erdős–Rényi–Gilbert model [16], [26], where each edge has independent probability p of being present. To assess scalability, we vary N and p in our experiments according to a parameter x where $N = 2^x$ and $p = 2^{2-x}$ for $x \in [4, 12]$: these choices lead to networks where 2–6% of nodes are disconnected from the others. We use BGP routes and a pure shortest-path policy based on SP for these networks, and check that nodes can reach the destination: we then report counterexamples for disconnected nodes and a positive verification result for connected nodes. Once again, we used a script to generate interfaces using a shortest paths algorithm. To choose cuts, we use a graph partitioning tool, hMETIS [35], to compute i fragments for each network. The computed fragments minimize the number of edges cut between fragments, and capture clustering behavior of the topology, while minimizing variance in fragment size. We consider $i = 8$ and $i = 16$, and a full cut ($i = |V|$).

We show the maximal SMT solve times for these benchmarks in Fig. 9 and NV times in Fig. 10.[10] Monolithic verification hits our Z3 timeout at $N = 256$; with 8 fragments, verification times out on the 8 times larger $N = 2048$ network.

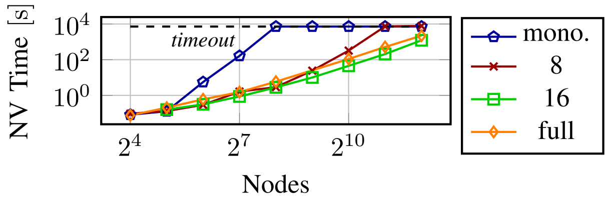


Fig. 10: NV times for random networks.

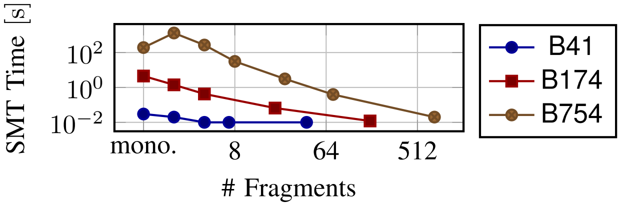


Fig. 11: Largest SMT solve times for TopologyZoo networks.

With 16 fragments, Z3 verifies $N = 4096$ in under 8 minutes, and when fully partitioning, Z3 takes at most 0.4 seconds to finish, with NV terminating for $N = 4096$ after 37 minutes.

Backbone Networks. We evaluated Kirigami with backbone network topologies from the Internet Topology Zoo [39]. We consider three networks: a 41-node (50-edge) topology B41, a 174-node (205-edge) topology B174 and a 754-node (895-edge) topology B754. As before, we model BGP routing throughout. B41's policy enforces no-transit and drops routes transiting [20] through AS customers or peers (relationships inferred from the topology [45]). B174 and B754 use a shortest-path policy as in SP. As with the random networks, we use hMETIS to compute i fragments. We consider $i = 2, 4, 7, 41$ for B41, $i = 2, 4, 20, 174$ for B174, and $i = 2, 4, 8, 25, 75, 754$ for B754.

[10] We included verification times for both connected and disconnected fragments: we did not see significant differences between the two.

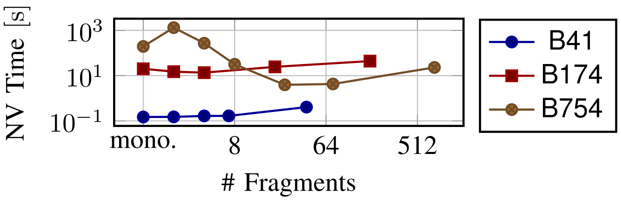


Fig. 12: NV times for TopologyZoo networks.

Fig. 11 and Fig. 12 show how the number of fragments affects SMT solve time and NV time, respectively. Like other benchmarks, larger cuts lead to greater reductions in SMT solve time, while NV time is lowest for non-full cuts ($i = 4$ for B174 and $i = 25$ for B754).

VIII. Related Work

Data Plane Analysis. Much prior work has analyzed properties of the network data plane [7], [34], [36]–[38], [43], [46], [49]. These tools operate on snapshots of the data plane — representing the global forwarding state at a single point in time — and verify that forwarding properties are satisfied.

Our approach most closely resembles the work of Jayaraman *et al.* on SecGuru and RCDC [34]. SecGuru verifies reachability using invariants it infers from specific data center topologies: our work develops a formal theory to verify arbitrary properties and invariants as specified by a user's interface, provides a framework for doing so automatically and instead focuses on the control plane.

Another relevant work is that of Plotkin *et al.* [49]. They demonstrate the use of bisimulations to relate simpler networks and formulas to more complex ones, improving verification scalability. Modular verification is recognized as a viable direction but left as future work; we focus on using modular verification in the control plane.

Control Plane Analysis. Our open SRP model builds on work on formal models of control planes, in particular, Bonsai's SRP model [10]. Unlike other prior work [13], [27], [28], we ignore network convergence and focus on networks with unique solutions, following other state-of-the-art [22], [44].

Two efforts closest to our own in modular control plane verification are Lightyear [56] and Timepiece [5]. Lightyear verifies safety properties for BGP networks using local invariant checks on a network's nodes and edges. Invariants in Lightyear are formulas rather than concrete routes, reducing the annotation burden; however, it cannot verify reachability properties and is restricted to BGP. Timepiece likewise chooses different tradeoffs to Kirigami: while it also allows users to specify arbitrary invariants, the user must provide specific times for their invariants and reason over when routes arrive, which may be difficult to do for complex networks.

Other control plane verification tools scale by abstracting routing behaviors, rather than modularizing the network. Bonsai [10] and Origami [24] *compress* concrete networks to smaller abstract networks which soundly approximate the original. Compression requires similar forwarding behavior between nodes; our approach avoids this restriction.

Our SMT encoding is inspired by Minesweeper [9], although we do not consider packet forwarding (only routing) and Minesweeper cannot perform modular verification. Several other tools [1], [11], [25], [44], [50], [58] use simulation-based techniques to scale verification up to large networks, but make pragmatic choices as to what arbitrary behaviors they can represent or properties they can verify. For instance, Plankton [50] uses explicit-state model checking to check a comparable set of properties to Minesweeper. Plankton can analyze networks with symbolic packets by using equivalence classes, but appears to need additional support to model other routing characteristics symbolically. None of these tools modularize the network: one could potentially extend them with modular techniques to improve their scalability. Other analyses also do not consider modularizing the network, and many are more restrictive than our approach: either limited to specific network properties [17], [22] or to specific protocols [57].

Modular Verification. Our work borrows from the compositional verification technique of *assume-guarantee reasoning* [6], [18], [23]. Such reasoning has been widely used in software, hardware and reactive systems [18], [29], [32]. While [42] applies assume-guarantee in network congestion control, it appears to be unexplored in analyzing routing. Instead of modeling processes, we model network fragments, whose shared environment is their input and output nodes. By requiring a partition's assumptions and guarantees to be equal, our reasoning avoids the common pitfall of circularity by relying on the stability of an open SRP's solution. Work exists on improving SMT solver performance by heuristically partitioning an SMT instance into independent instances with distinct search spaces [33]. Our approach is specifically for network fragments, where we focus on *generating* already-partitioned SMT formulas. For large formulas, we could additionally apply formula partitioning techniques.

IX. Conclusion

Networks are growing faster than SMT-based verification can scale. Scalable and modular verification techniques can harness the fact that operators build networks bottom-up using local policies at each node. By providing interfaces describing a network's local invariants, operators can make their networks more robust and easier to understand. We present a formal model representing a network as a collection of fragments, and show how our modular verification procedure uses this model to catch bugs and check the correctness of users' interfaces. We prove our procedure is sound and demonstrate it with Kirigami, which dramatically speeds up network verification.

X. Acknowledgments

We thank the ICNP anonymous reviewers and our shepherd, Olaf Maennel, for their astute and helpful feedback.

References

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

[2] Anubhavnidhi Abhashkumar, Kausik Subramanian, Alexey Andreyev, Hyojeong Kim, Nanda Kishore Salem, Jingyi Yang, Petr Lapukhov, Aditya Akella, and Hongyi Zeng. Running BGP in data centers at scale. In *18th USENIX Symposium on Networked Systems Design and Implementation, NSDI*, pages 65–81. USENIX Association, 2021.

[3] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. In *SIGCOMM*, 2008.

[4] Timothy Alberdingk Thijm, Ryan Beckett, Aarti Gupta, and David Walker. Kirigami, the verifiable art of network cutting, 2022. https://arxiv.org/abs/2202.06098.

[5] Timothy Alberdingk Thijm, Ryan Beckett, Aarti Gupta, and David Walker. Modular control plane verification via temporal invariants, 2022. https://arxiv.org/abs/2204.10303.

[6] Rajeev Alur and Thomas A Henzinger. Reactive modules. *Formal methods in system design*, 15(1):7–48, 1999.

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

[8] Clark Barrett and Cesare Tinelli. Satisfiability modulo theories. In *Handbook of model checking*, pages 305–343. Springer, 2018.

[9] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. A general approach to network configuration verification. In *SIGCOMM*, August 2017.

[10] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. Control plane compression. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, pages 476–489, New York, NY, USA, 2018. ACM.

[11] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. Abstract interpretation of distributed network control planes. *Proceedings of the ACM on Programming Languages*, 4(POPL):1–27, 2019.

[12] R. Chandra, P. Traina, and T. Li. BGP communities attribute. rfc 1997, RFC Editor, 1996. https://www.rfc-editor.org/rfc/rfc1997.txt.

[13] Matthew L Daggitt, Alexander JT Gurney, and Timothy G Griffin. 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*, pages 103–116. ACM, 2018.

[14] M. Danelutto and R. Di Cosmo. A "minimal disruption" skeleton experiment: Seamless map & reduce embedding in ocaml. *Procedia Computer Science*, 9:1837–1846, 2012. Proceedings of the International Conference on Computational Science, ICCS 2012.

[15] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *TACAS*, March 2008.

[16] Paul Erdös and Alfréd Rényi. On random graphs i. *Publicationes Mathematicae (Debrecen)*, 6:290–297, 1959. https://www.renyi.hu/~p_erdos/1959-11.pdf.

[17] Seyed K. Fayaz, Tushar Sharma, Ari Fogel, Ratul Mahajan, Todd Millstein, Vyas Sekar, and George Varghese. Efficient network reachability analysis using a succinct control plane representation. In *OSDI*, 2016. https://www.usenix.org/system/files/conference/osdi16/osdi16-fayaz.pdf.

[18] Cormac Flanagan and Shaz Qadeer. Thread-modular model checking. In *International SPIN Workshop on Model Checking of Software*, pages 213–224. Springer, 2003.

[19] Ari Fogel, Stanley Fung, Luis Pedrosa, Meg Walraed-Sullivan, Ramesh Govindan, Ratul Mahajan, and Todd Millstein. A general approach to network configuration analysis. In *NSDI*, October 2015. https://www.usenix.org/system/files/conference/nsdi15/nsdi15-paper-fogel.pdf.

[20] Lixin Gao. On inferring autonomous system relationships in the internet. *IEEE/ACM Transactions on networking*, 9(6):733–745, 2001. https://ieeexplore.ieee.org/abstract/document/974527.

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

[22] Aaron Gember-Jacobson, Raajay Viswanathan, Aditya Akella, and Ratul Mahajan. Fast control plane analysis using an abstract representation. In *SIGCOMM*, August 2016.

[23] Dimitra Giannakopoulou, Kedar S Namjoshi, and Corina S Păsăreanu. Compositional reasoning. In *Handbook of Model Checking*, pages 345–383. Springer, 2018.

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

[25] Nick Giannarakis, Devon Loehr, Ryan Beckett, and David Walker. NV: An intermediate language for verification of network control planes. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 958–973, New York, NY, USA, 2020. Association for Computing Machinery.

[26] Edgar N Gilbert. Random graphs. *The Annals of Mathematical Statistics*, 30(4):1141–1144, 1959. https://www.jstor.org/stable/2237458.

[27] Timothy G. Griffin, F. Bruce Shepherd, and Gordon Wilfong. The stable paths problem and interdomain routing. *IEEE/ACM Trans. Networking*, 10(2), 2002. https://ieeexplore.ieee.org/abstract/document/993304.

[28] Timothy G. Griffin and Joäo Luís Sobrinho. Metarouting. In *SIGCOMM*, pages 1–12, August 2005.

[29] Orna Grumberg and David E Long. Model checking and modular verification. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(3):843–871, 1994.

[30] Chuanxiong Guo, Guohan Lu, Dan Li, Haitao Wu, Xuan Zhang, Yunfeng Shi, Chen Tian, Yongguang Zhang, and Songwu Lu. BCube: A high performance, server-centric network architecture for modular data centers. In *SIGCOMM*, 2009.

[31] Chuanxiong Guo, Haitao Wu, Kun Tan, Lei Shi, Yongguang Zhang, and Songwu Lu. Dcell: A scalable and fault-tolerant network structure for data centers. In *SIGCOMM*, 2008.

[32] Thomas A Henzinger, Shaz Qadeer, and Sriram K Rajamani. You assume, we guarantee: Methodology and case studies. In *International Conference on Computer Aided Verification*, pages 440–451. Springer, 1998.

[33] Antti EJ Hyvärinen, Matteo Marescotti, and Natasha Sharygina. Search-space partitioning for parallelizing smt solvers. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 369–386. Springer, 2015.

[34] Karthick Jayaraman, Nikolaj Bjørner, Jitu Padhye, Amar Agrawal, Ashish Bhargava, Paul-Andre C Bissonnette, Shane Foster, Andrew Helwer, Mark Kasten, Ivan Lee, Anup Namdhari, Haseeb Niaz, Aniruddha Parkhi, Hanukumar Pinnamraju, Adrian Power, Neha Milind Raje, and Parag Sharma. Validating datacenters at scale. In *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM '19, page 200–213, New York, NY, USA, 2019. Association for Computing Machinery.

[35] G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar. Multilevel hypergraph partitioning: applications in VLSI domain. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 7(1):69–79, 1999. https://ieeexplore.ieee.org/abstract/document/748202.

[36] Peyman Kazemian, Michael Chang, Hongyi Zeng, George Varghese, Nick McKeown, and Scott Whyte. Real time network policy checking using header space analysis. In *NSDI*, pages 99–112, April 2013. https://www.usenix.org/system/files/conference/nsdi13/nsdi13-final8.pdf.

[37] Peyman Kazemian, George Varghese, and Nick McKeown. Header space analysis: Static checking for networks. In *NSDI*, April 2012. https://www.usenix.org/system/files/conference/nsdi12/nsdi12-final8.pdf.

[38] Ahmed Khurshid, Xuan Zou, Wenxuan Zhou, Matthew Caesar, and P. Brighten Godfrey. Veriflow: Verifying network-wide invariants in real time. In *NSDI*, April 2013. https://www.usenix.org/system/files/conference/nsdi13/nsdi13-final100.pdf.

[39] Simon Knight, Hung X. Nguyen, Nickolas Falkner, Rhys Bowden, and Matthew Roughan. The internet topology zoo. *IEEE Journal on Selected Areas in Communications*, 29(9):1765–1775, 2011. https://ieeexplore.ieee.org/abstract/document/6027859.

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

[41] K Rustan M Leino. Dafny: An automatic program verifier for functional correctness. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, pages 348–370. Springer, 2010.

[42] Alessio Lomuscio, Ben Strulo, Nigel Walker, and Peng Wu. Assume-guarantee reasoning with local specifications. In *International conference on formal engineering methods*, pages 204–219. Springer, 2010.

[43] Nuno P. Lopes, Nikolaj Bjørner, Patrice Godefroid, Karthick Jayaraman, and George Varghese. Checking beliefs in dynamic networks. In *NSDI*, 2015. https://www.usenix.org/system/files/conference/nsdi15/nsdi15-paper-lopes.pdf.

[44] Nuno P Lopes and Andrey Rybalchenko. Fast BGP simulation of large datacenters. In *International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 386–408. Springer, 2019. https://web.ist.utl.pt/nuno.lopes/pubs/fastplane-vmcai19.pdf.

[45] M. Luckie, B. Huffaker, k. claffy, A. Dhamdhere, and V. Giotsas. As relationships, customer cones, and validation. In *ACM Internet Measurement Conference (IMC)*, pages 243–256, 2013-10.

[46] Haohui Mai, Ahmed Khurshid, Rachit Agarwal, Matthew Caesar, P. Brighten Godfrey, and Samuel Talmadge King. Debugging the data plane with anteater. In *SIGCOMM*, 2011.

[47] Kieren McCarthy. BGP super-blunder: How verizon today sparked a 'cascading catastrophic failure' that knackered cloudflare, amazon, etc. https://www.theregister.com/2019/06/24/verizon_bgp_misconfiguration_cloudflare/, 2019.

[48] Ivan Pepelnjak. Valley-free routing in data center fabrics. https://blog.ipspace.net/2018/09/valley-free-routing-in-data-center.html, 2018.

[49] Gordon D. Plotkin, Nikolaj Bjørner, Nuno P. Lopes, Andrey Rybalchenko, and George Varghese. Scaling network verification using symmetry and surgery. In *POPL*, January 2016.

[50] Santhosh Prabhu, Ali Kheradmand, Brighten Godfrey, and Matthew Caesar. Predicting network futures with plankton. In *Proceedings of the First Asia-Pacific Workshop on Networking*, APNet'17, pages 92–98, August 2017.

[51] Yakov Rekhter, Tony Li, Susan Hares, et al. A border gateway protocol 4 (BGP-4). RFC 4271, RFC Editor, 2006. https://www.rfc-editor.org/rfc/rfc4271.txt.

[52] Simon Sharwood. Facebook rendered spineless by buggy audit code that missed catastrophic network config error. https://www.theregister.com/2021/10/06/facebook_outage_explained_in_detail/, 2021.

[53] João Luís Sobrinho. An algebraic theory of dynamic network routing. *IEEE/ACM Trans. Netw.*, 13(5):1160–1173, October 2005. https://ieeexplore.ieee.org/abstract/document/1528502.

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

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

[56] Alan Tang, Ryan Beckett, Karthick Jayaraman, Todd Millstein, and George Varghese. LIGHTYEAR: Using modularity to scale BGP control plane verification, 2022. https://arxiv.org/abs/2204.09635.

[57] Konstantin Weitz, Doug Woos, Emina Torlak, Michael D. Ernst, Arvind Krishnamurthy, and Zachary Tatlock. Formal semantics and automated verification for the border gateway protocol. In *NetPL*, March 2016. https://www.dougwoos.com/papers/bagpipe-netpl16.pdf.

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

[59] Jin Y Yen. Finding the k shortest loopless paths in a network. *Management Science*, 17(11):712–716, 1971.