KATRA: Realtime Verification for Multilayer Networks

Ryan Beckett, *Microsoft;* Aarti Gupta, *Princeton University*

https://www.usenix.org/conference/nsdi22/presentation/beckett

This paper is included in the Proceedings of the 19th USENIX Symposium on Networked Systems Design and Implementation.

April 4–6, 2022 • Renton, WA, USA

978-1-939133-27-4

Katra: Realtime Verification for Multilayer Networks

Ryan Beckett
Microsoft

Aarti Gupta
Princeton University

Abstract

We present a new verification algorithm to efficiently and incrementally verify arbitrarily *layered* network data planes that are implemented using packet encapsulation. Inspired by work on model checking of pushdown systems for recursive programs, we develop a verification algorithm for networks with packets consisting of *stacks* of headers. Our algorithm is based on a new technique that lazily "repairs" a decomposed stack of header sets on demand to account for cross-layer dependencies. We demonstrate how to integrate our approach with existing fast incremental data plane verifiers and have implemented our ideas in a tool called Katra. Evaluating Katra against an alternative approach based on equipping existing incremental verifiers to emulate finite header stacks, we show that Katra is between 5x-32x faster for packets with just 2 headers (layers), and that its performance advantage grows with both network size and layering.

1 Introduction

The success of networks is in part due to their *layered* design where different protocol layers are delegated different responsibilities. For instance, many networks, including virtual networks [3, 16, 31], are designed in an overlay/underlay pattern that is implemented by encapsulating packets, e.g. using IP-in-IP [34], IP GRE [18], or VXLAN [30] tunnels. In a wide-area network (WAN), routing protocols such as iBGP and SDN solutions such as SWAN [19] rely on an underlying label-switching protocol like MPLS. In routers, Ethernet frames are encapsulated in IP headers to implement forwarding, and new and emerging technologies such as SD-WAN can connect two WANs together by tunneling packets to each other securely using IPSEC [28].

However, while this layered design has proven successful, allowing each layer to hide many details from others, it also makes operating networks reliably a challenge as many bugs can sit in the intersection of one or more of these layers [37]. Given the pervasiveness of layering as a fundamental design pattern in networks, it is critical that we be able to ensure the reliability of networks using this design. In practice, the implementation of layering is often extraordinarily complex. For instance, packets going over AT&T's backbone network consist of as many as eleven encapsulated headers [44].

To ensure the safe operation of multilayer networks, a natural technology to employ is that of network verification, which has emerged as a viable technique to proactively catch bugs and misconfigurations related to automation and human error. Employed by nearly all major cloud providers now [5, 17, 21, 39, 45], network verification has witnessed substantial practical use in industry as researchers have iteratively improved upon the scalability, responsiveness, and expressiveness of the underlying verification tools and techniques. Many of these tools are engineering marvels – through complex data structures and algorithms, they enable efficient verification of new network changes in milliseconds.

While recent progress in network verification has been substantial, existing tools typically analyze a single layer or component of the network stack. For example, most verification tools in use today analyze only the simple IP-based forwarding tables and ACLs [4, 7, 20, 21, 26, 27, 29, 41, 42, 46] governed by the control plane. To verify a network with N layers using verification tools today, one possibility is to model packets in the network as consisting of N duplicate copies of different headers (e.g., an IPv4 header). While this approach is possible, and researchers have proposed this approach in some prior work [42, 46], it suffers from two major problems.

The first problem is that the number of layers N may not be known a priori by the user of the verification tool. For instance, a network making use of the MPLS fast reroute (FRR) [14, 25, 33] protects links against failure by rerouting traffic that would have gone over the failed link along a predetermined backup tunnel. FRR schemes may encapsulate a packet's header a number of times that depends on the number of failures encountered by the packet along its path. Moreover, if a user provides an incorrect (low) estimate of N, then the verifier may report both false positives and false negatives.

The second problem is that, even when the user is able to statically determine N, modeling this many packet headers simultaneously leads to a highly inefficient representation. Most network verifiers work by modeling the sets of packets that are reachable from each ingress and egress point in the network [27, 41, 42, 46], and such sets may grow substantially larger and more complex to process when capturing the dependencies between different headers in the packet. In our experiments (§7), even with just two layers, this simple approach can be anywhere from 5x-32x slower than necessary.

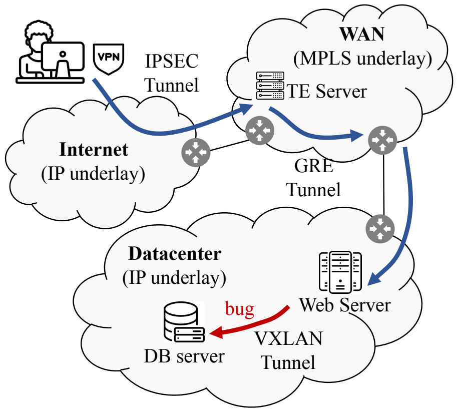


Figure 1: A user attempts to access a corporate web service.

To address these problems, in this paper, we present Katra, the first tool for real-time verification of layered networks – those networks that manipulate *stacks* of headers through packet encapsulation. Inspired by recent work on analyzing MPLS [22, 23], we leverage ideas from the verification of pushdown systems [36] used to model recursive programs. A key new idea is to keep a decomposed representation of the set of header stacks as a stack of header sets where the set of headers at one layer in the stack are treated as independent of those in another. When an operation modifies and later restricts packets at a given layer, it may be necessary to go back and "repair" the sets of packets representative of previous headers in the stack to account for this change.

We demonstrate how we can integrate our ideas with existing incremental data plane verifiers, and we implement Katra as an extension to the state-of-the-art verifier APKeep [46]. In addition to being able to reason about arbitrarily large header stacks, for networks consisting of just two layers, we further demonstrate that Katra can verify properties 5x-32x faster than an approach based on extending existing algorithms with a finite header stack encoding.

Our contributions with Katra are the following:

- We present a new formal model, and its semantics, for layered networks that supports arbitrarily nested packet encapsulations and decapsulations.
- We develop an efficient algorithm for verifying layered networks in our model. The algorithm is based on a new notion of *partial* equivalence class as well as a new decomposed representation for symbolic header stacks.
- We implement our ideas in a tool called Katra, which integrates our new model and algorithm with the state-of-the-art incremental verification approach based on APKeep [46].
- We evaluate Katra against a baseline based on a finite header stack representation and demonstrate that Katra is 5x-32x faster with this speedup growing larger with both network size and the number of network layers.

Networking example	**Layering mechanism**
campus network isolation [44]	VLAN, VXLAN [30]
virtual private network [15]	IP-IP [34], IP GRE [18]
3G/4G mobile packet core [38]	GPRS tunneling [1]
interior gateway protocol [32]	LSPs, MPLS [35]
performance proxy [8]	IPSEC [28]
traffic engineering [43]	IP GRE [18]
software-defined WAN [13]	IPSEC [28]

Table 1: Example network services features with layering.

2 Motivation and Background

Protocol *layering* in networks is used pervasively as a way to separate concerns and build new features and services atop existing infrastructure. Consider the scenario depicted in Figure 1. In the example, an employee of a company attempts to access a corporate web service, that is hosted in the cloud, from their home. The employee uses a secure VPN connection so their traffic is encapsulated in an IPSEC [28] tunnel before being forwarded over the Internet. When the traffic traverses the cloud provider's wide area network (WAN), a traffic engineering server selects an egress point for a nearby data center. The WAN forwards packets to this egress using an IP-GRE tunnel over its MPLS-based core. Once the traffic reaches the data center, it is forwarded to the requested web server. The web server must now access data from a database (DB) server configured in the same virtual overlay network. The web server thus sends traffic to the DB server using a VXLAN tunnel configured atop an IP-based datacenter fabric.

In the example, layer 2 and layer 3 forwarding elements are combined to implement several abstractions. While this approach to protocol layering is powerful, if any one of the forwarding policies at any layer in the example is misconfigured, then the user will not be able to reach the intended service. For instance, if there were a misconfigured security rule in the overlay network between the web and DB servers then the user may lose connectivity to the web service. Similarly, a misconfiguration in the datacenter's IP-based underlay network could break connectivity. To make matters worse, when network forwarding bugs span multiple layers, identifying the root cause of the issue can be complicated as it may require coordination between multiple different teams spread across one or more organizations [37], each with only a partial view of the network as a whole.

Multilayer verification. Network verification is a natural fit to ensure the correctness of multilayer networks, yet verifiers today were not built with layering in mind. Existing verifiers assume that packets have a fixed size header rather than an expandable *stack* of headers. While it is sometimes possible to retroactively analyze such multilayer networks using existing verifiers by pessimistically modeling a "worst case" fixed size header stack, doing so is often highly inefficient.

3 Layered Network Model

Rather than model networks with fixed size headers, as is done by existing network verifiers, in this section we define a new network model that includes protocol layering as a first class concern. We model multilayer networks as operating on an unbounded *stack* of headers. We then demonstrate that one can view both single-layer and multi-layer networks as instances of our general model.

3.1 Notation and Preliminaries

Before defining our network model, we first introduce some notation and preliminary definitions.

Definition 3.1 (Sequences). *For a set X, we use the notation X^* to mean the set of all possible sequences of elements of X. We define a sequence $\sigma \in X^*$ inductively as either ε representing the empty sequence, or the concatenation $(\sigma' \cdot x)$ of another sequence $\sigma' \in X^*$ together with an element $x \in X$.*

For simplicity of notation, we sometimes write a sequence $\sigma \in X^*$ as $\sigma = x_0 \cdot x_1 \cdot \cdots \cdot x_n$ where $x_i \in X$ and x_0 is the first element of the sequence and x_n is the last element, and we omit writing out the ε. Concatenation of two sequences is defined recursively as $\sigma \cdot \varepsilon = \sigma$ and $\sigma \cdot (\sigma' \cdot x) = (\sigma \cdot \sigma') \cdot x$ As a shorthand, for a sequence $\sigma = \sigma' \cdot x$ we define $\text{top}(\sigma) = x$ and $\text{bot}(\sigma) = \sigma'$. These two partial functions are undefined when $\sigma = \varepsilon$. Finally, we write $|\sigma|$ to mean the length of a sequence such that $|\varepsilon| = 0$ and $|\sigma \cdot x| = 1 + |\sigma|$. A stack is a sequence σ where the top of the stack is given by $\text{top}(\sigma)$.

For sets X and Y, we use the standard notation Y^X or $X \rightarrow Y$ to mean the set of functions from X to Y. Similarly, we use the notation $X \hookrightarrow Y$ to represent partial functions from X to Y. Given a (potentially partial) function f from X to Y and function g from Y to Z, we write $f \circ g$ to define their composition, from X to Z.

3.2 Formal network model

We define a network $\mathcal{N}$ as a tuple $\langle V, E, \mathcal{H}, \mathcal{T}, \mathcal{R} \rangle$ where:

- V is a set of vertices, and $E \subseteq 2^{V \times V}$ is a set of edges. Edges are unidirectional and we represent bidirectional edges with a pair of edges. For edge $e = \langle u, v \rangle$, we use the notation $\text{src}(e) = u$ and $\text{tgt}(e) = v$.
- $\mathcal{H}$ is a set of valid headers for the protocols in use.
- $\mathcal{T}$ is a set of transformations, which are partial functions over packet headers of type $\mathcal{T} \subseteq 2^{\mathcal{H} \hookrightarrow \mathcal{H}}$.
- $\mathcal{R}$ is a set of rules $\mathcal{R} \subseteq \mathbb{N} \times E \times 2^{\mathcal{H}} \times \mathcal{T}$. For rule $r \in \mathcal{R}$ where $r = \langle p, e, m, \tau \rangle$. We use the notation $\text{priority}(r) = p$, $\text{edge}(r) = e$, $\text{match}(r) = m$, and $\text{modify}(r) = \tau$ to refer to the components of the rule.

Intuitively, lower priority rules take precedence over those with higher priority at a given node. The best rule for a given header at a node is the rule with lowest priority at that node that also matches the header.

Definition 3.2 (Best rule). *For a node $u \in V$ and a header $h \in \mathcal{H}$, we define the best matching rule at u as: $\Omega(u,h) = \min_{\text{priority}}\{r \in \mathcal{R} \mid \text{src}(\text{edge}(r)) = u,\ h \in \text{match}(r)\}$*

For simplicity, we assume that there is always a unique best rule for a given header. This is ensured by requiring that (1) rule priorities must be unique, and (2) all headers are matched by at least one rule. In practice, one can ensure this requirement is met by adding a maximum priority default rule that matches all other unmatched headers.

3.3 Network semantics

Given a header $h \in \mathcal{H}$ and an initial node $u \in V$, which we call a located packet $\mathcal{L} = V \times \mathcal{H}$, the network produces a sequence of new located packets to capture the packet's history as it goes through the network. Specifically, we define the semantics of a network $\mathcal{N}$ as a function $\llbracket \mathcal{N} \rrbracket_i : \mathcal{L} \rightarrow \mathcal{L}^*$ that takes an initial located packet to a trace of located packets through the network for a given number of steps $i \in \mathbb{N}$:

$$\llbracket \mathcal{N} \rrbracket_i \langle u, h \rangle = \begin{cases} \varepsilon \cdot \langle u, h \rangle & \text{if } i = 0 \\ \sigma & \text{elif } \tau(h') \text{ undefined} \\ \sigma \cdot \langle v, \tau(h') \rangle & \text{otherwise} \end{cases}$$

where $\sigma = \llbracket \mathcal{N} \rrbracket_{i-1} \langle u, h \rangle$ and $\text{top}(\sigma) = \langle u', h' \rangle$ and:

$$\begin{aligned} \tau &= \text{modify}(\Omega(u', h')) \\ v &= \text{tgt}(\text{edge}(\Omega(u', h'))) \end{aligned}$$

Definition 3.3 (Packet termination). *We say network $\mathcal{N}$ has terminated a located packet ℓ after i steps, written $\mathcal{N} \otimes \langle i, \ell \rangle$, if the trace no longer changes: $\llbracket \mathcal{N} \rrbracket_i \ell = \llbracket \mathcal{N} \rrbracket_{i-1} \ell$.*

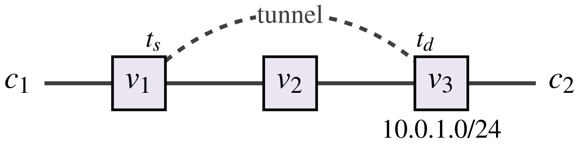
3.4 Lifting networks to layered networks

To implement layering, conceptually packets contain an unbounded stack of headers to which the network can push or pop. A layered network is just an instance of a network that processes *stacks* of headers:

Definition 3.4 (Multilayer network). *A multilayer network is an instance of a network $\mathcal{N} = \langle V, E, \mathcal{H}^*, \mathcal{T}, \mathcal{R} \rangle$ over sequences of headers $\mathcal{H}^*$ with some restrictions on $\mathcal{T}$ and $\mathcal{R}$.*

Primarily, we require that every transformation $\tau \in \mathcal{T}$ and every rule match set $\text{match}(r)$ for $r \in \mathcal{R}$ only inspects or modifies the top of the stack[1]. In other words, we often write that $\tau(\sigma \cdot x) = \sigma \cdot \tau(x)$. And that $\sigma \cdot h \in \text{match}(r) \iff h \in \text{match}(r)$ as though τ and $\text{match}(r)$ were defined over $\mathcal{H}$. The

[1] This restriction can express encapsulation and decapsulation in real networks yet also makes verification tractable.



rule	p	e	m	τ
r_0	100	$\langle c_1, v_1\rangle$	$\mathcal{H}^*$	τ_{id}
r_1	100	$\langle v_1, v_1\rangle$	$\phi_{\text{dst}}(23.1.4.0/24)$	$\tau_{\text{push}} \circ \tau_{\text{tunl}}$
r_2	200	$\langle v_1, v_2\rangle$	$\phi_{\text{dst}}(10.0.1.0/24)$	τ_{id}
r_3	300	$\langle v_1, v_1\rangle$	$\mathcal{H}^*$	τ_{drop}
r_4	100	$\langle v_2, v_3\rangle$	$\phi_{\text{dst}}(10.0.1.0/24)$	τ_{id}
r_5	200	$\langle v_2, v_2\rangle$	$\mathcal{H}^*$	τ_{drop}
r_6	100	$\langle v_3, v_3\rangle$	$\phi_{\text{src}}(t_s) \cap \phi_{\text{dst}}(t_d)$	τ_{pop}
r_7	200	$\langle v_3, c_2\rangle$	$\mathcal{H}^*$	τ_{id}
r_8	100	$\langle c_2, c_2\rangle$	$\mathcal{H}^*$	τ_{delv}

Figure 2: Example formulation of a network with a single tunnel between v_1 and v_3. We use the notation $\phi_f(P)$ for set P as a shorthand to mean packets where the field f is contained in P. Thus $\phi_{\text{dst}}(P) = \{\sigma \in \mathcal{H}^* \mid \langle d, s\rangle \in \text{top}(\sigma), d \in P\}$. Tunneled packets are encapsulated by first executing τ_{push} to duplicate the top-most header before modifying this header copy with τ_{tunl} to set to source ip to $t_s = 10.0.2.0$ and the destination ip to $t_d = 10.0.1.0$. Packets are then forwarded according to the underlay network towards $t_d \in 10.0.1.0/24$ hosted at v_3. Packets at v_3 are decapsulated by popping the top-most header and then delivered to client c_2.

i	$\text{top}(\llbracket \mathcal{N} \rrbracket_i \langle c_1, \langle d, s\rangle\rangle)$	description
0	$\langle c_1, \langle d, s\rangle\rangle$	*forward to v_1*
1	$\langle v_1, \langle d, s\rangle\rangle$	*encapsulate*
2	$\langle v_1, \langle d, s\rangle \cdot \langle t_s, t_d\rangle\rangle$	*forward to v_2*
3	$\langle v_2, \langle d, s\rangle \cdot \langle t_s, t_d\rangle\rangle$	*forward to v_3*
4	$\langle v_3, \langle d, s\rangle \cdot \langle t_s, t_d\rangle\rangle$	*decapsulate*
5	$\langle v_3, \langle d, s\rangle\rangle$	*forward to c_2*
≥ 6	$\langle c_2, \langle d, s\rangle\rangle$	*delivered*

Table 2: Trace of a packet with source s and destination d from client node c_1 in the network shown in Figure 2.

only exception to this restriction is for two special transformations τ_{push}, which makes a new copy of the current top of the stack, and τ_{pop}, which drops or decapsulates the top of the stack. More specifically, we define $\tau_{\text{push}}(\sigma \cdot x) = \sigma \cdot x \cdot x$ and we define $\tau_{\text{pop}}(\sigma \cdot x \cdot y) = \sigma \cdot x$. Both transformations are undefined otherwise and may be composed with other transformations (e.g., $\tau \circ \tau_{\text{push}}$).

Example Network. To make the model more concrete, we show an example of its instantiation in Figure 2. The network in the figure is given by the tuple $\mathcal{N} = \langle V, E, \mathcal{H}^*, \mathcal{T}, \mathcal{R}\rangle$ where the nodes and edges are defined by the sets:

$$\begin{aligned} V &= \{c_1, v_1, v_2, v_3, c_2\} \\ E &= \{\langle c_1, v_1\rangle, \langle v_1, v_1\rangle, \langle v_1, v_2\rangle, \langle v_2, v_3\rangle, \langle v_3, v_3\rangle, \langle v_3, c_2\rangle\} \end{aligned}$$

Note that the edges include self edges (e.g., $\langle v_1, v_1\rangle$) to model recursive lookup for forwarding and tunneling.

The set of headers $\mathcal{H} = \{\langle d, s\rangle \mid d, s \in \{0, \ldots, 2^{32} - 1\}\}$ defines headers consisting of two 32-bit IP address fields for destination and source IP, and $\mathcal{H}^*$ is all sequences (stacks) of such headers. The set of transformations for the network is given by $\mathcal{T} = \{\tau_{\text{id}}, \tau_{\text{drop}}, \tau_{\text{delv}}, \tau_{\text{push}} \circ \tau_{\text{tunl}}, \tau_{\text{pop}}\}$. The transformation τ_{id} is the identity transformation such that $\tau_{\text{id}}(\sigma) = \sigma$, τ_{drop} and τ_{delv} are transformations that are undefined for all inputs and thus terminate traffic, τ_{tunl} is a transformation that rewrites the source IP address to t_s and the destination IP address to t_d. The composed transformation $\tau_{\text{push}} \circ \tau_{\text{tunl}}$ first creates a copy of the top of the stack and then rewrites the IP addresses according to τ_{tunl} for the encapsulated header.

A trace through the network given by the semantics (see §3.3) represents the packet forwarding in the network. Consider sending an initial packet from client c_1 with some destination address $d \in \phi_{\text{dst}}(23.1.4.0/24)$ and some arbitrary source address s. The top of the trace given by the semantics $\llbracket \mathcal{N} \rrbracket_i \langle c_1, \langle d, s\rangle\rangle$ is shown in Table 2.

4 Realtime Verification of Layered Networks

Given a network $\mathcal{N} = \langle V, E, \mathcal{H}^*, \mathcal{T}, \mathcal{R}\rangle$ over stacks of headers $\mathcal{H}^*$ and a new rule r being inserted or removed from $\mathcal{R}$, our goal is to incrementally verify the correctness of $\mathcal{N}$ with respect to some user defined properties of interest.

In this section we first give a brief overview of how existing incremental verification algorithms work for finite header sets $\mathcal{H}$ (§4.1). We then show how this notion of equivalence class falls apart for the infinite space of header stacks $\mathcal{H}^*$, which leads existing algorithms to not terminate. To solve this problem, we define a new notion of *partial* equivalence class based on only the top of the header stack (§4.2). Partial equivalence classes can be computed efficiently, however they do not necessarily guarantee equivalent network-wide behavior. Instead, we develop an algorithm that lazily refines these classes while verifying properties (§4.3 and §4.4).

4.1 Existing incremental verifiers

Most incremental data plane verifiers today work by analyzing the network rules $\mathcal{R}$ and, based on that analysis, dividing the headers $\mathcal{H}$ into subsets that have the same forwarding behavior, which can then be checked using graph algorithms. More specifically equivalence classes are defined as:

Definition 4.1 (Trace hops). *Given a trace σ (from §3.3) consisting of located packets $\mathcal{L}$, we define function* $\text{hops}(\sigma)$, *which produces only the nodes in the trace, inductively over σ as* $\text{hops}(\varepsilon) = \varepsilon$ *and* $\text{hops}(\sigma' \cdot \langle u, h\rangle) = \text{hops}(\sigma') \cdot u$.

Definition 4.2 (Equivalence classes). *A set of header sets $\{\mathcal{H}_1,\dots,\mathcal{H}_n\}$ are equivalence classes for a network $\mathcal{N} = \langle V,E,\mathcal{H},\mathcal{T},\mathcal{R}\rangle$ if the following conditions hold:*

- $\mathcal{H} = \mathcal{H}_1 \cup \ldots \cup \mathcal{H}_n$ *(complete)*
- $\forall i,j \in \{1,..,n\},\ i \neq j \Rightarrow \mathcal{H}_i \cap \mathcal{H}_j = \emptyset$ *(disjoint)*
- $\forall j \in \{1,..,n\},\ \forall h_1,h_2 \in \mathcal{H}_j,\ \forall u \in V,\ \forall i \in \mathbb{N},$ *(g-equiv)*
  $\mathrm{hops}([\![\mathcal{N}]\!]_i\langle u,h_1\rangle) = \mathrm{hops}([\![\mathcal{N}]\!]_i\langle u,h_2\rangle)$

Existing incremental verification tools compute an over-approximate set of equivalence classes $\{\mathcal{H}_1,\dots,\mathcal{H}_n\}$ using intricate data structures such as multi-dimentional tries [29] and Binary Decision Diagrams (BDDs) [46]. While early work on incremental verification such as Veriflow [29] and Deltanet [20] could not handle rule transformations (i.e., all transformations must be τ_{id}), more recent work such as AP [42] and APKeep [46] can account for transformations.

At a high-level, these tools work as follows. First, they compute a set of equivalence classes $\{\mathcal{H}_1,\dots,\mathcal{H}_n\}$ based on the rule match fields. Next, for each transformation $\tau \in \mathcal{T}$ and each $i \in \{1,..,n\}$, they compute $\tau(\mathcal{H}_i) = \{\tau(h) \mid h \in \mathcal{H}_i\}$. Since the transformed sets may now violate the disjoint condition, the resulting set $\{\mathcal{H}_1,\dots,\mathcal{H}_n,\tau(\mathcal{H}_1),\dots,\tau(\mathcal{H}_n)\}$ is made disjoint by dividing up these sets. This process is iterated with all transformations until no more changes occur.

The problem with equivalence classes. There are several problems that occur when trying to lift this approach to equivalence class generation to *stacks* of headers. One problem is that the space of header stacks $\mathcal{H}^*$ is infinite and symbolic data structures in existing tools cannot represent and manipulate infinite sets of values. This is generally a hard problem, which we solve in §4.3.

Even with data structures to manipulate such infinite sets, the algorithm discussed previously does not necessarily terminate in this infinite space. For instance, an equivalence class for stacks with a single header: $\mathcal{H}_i^* = \{\varepsilon \cdot h \mid h \in \mathcal{H}\}$ does not terminate with τ_{push} – one would compute a new equivalence class for packets with two headers, then three, and so on.

4.2 Partial equivalence classes

To solve this problem, we introduce a new notion of *partial* equivalence classes. Partial equivalence classes capture sets of packets that will have the same forwarding behavior at every node in the network but may not be transformed unambiguously by transformations to other partial equivalence classes. Formally, we define them as:

Definition 4.3 (Partial Equivalence Classes). *A set of header sets $\{\mathcal{H}_1,\dots,\mathcal{H}_n\}$ are partial equivalences classes for a network $\mathcal{N} = \langle V,E,\mathcal{H},\mathcal{T},\mathcal{R}\rangle$ if the following hold:*

- $\mathcal{H} = \mathcal{H}_1 \cup \ldots \cup \mathcal{H}_n$ *(complete)*
- $\forall i,j \in \{1,..,n\},\ i \neq j \Rightarrow \mathcal{H}_i \cap \mathcal{H}_j = \emptyset$ *(disjoint)*
- $\forall j \in \{1,..,n\},\ \forall h_1,h_2 \in \mathcal{H}_j,\ \forall u \in V,$ *(l-equiv)*
  $\mathrm{edge}(\Omega(u,h_1)) = \mathrm{edge}(\Omega(u,h_2)) \wedge$
  $\mathrm{modify}(\Omega(u,h_1)) = \mathrm{modify}(\Omega(u,h_2))$

The difference between partial equivalence classes and equivalence classes (Definition 4.2) is subtle. We demonstrate the difference in Figure 3. In the example, packet headers consist of a destination IP field and time-to-live (TTL) field. If we ignore the layering transformations τ_{push} and τ_{pop}, which make the example not terminate, existing tools AP and APKeep would compute the equivalence classes shown in Figure 3b according to Definition 4.2. There are 257 equivalence classes. This large number comes from repeatedly applying τ_{ttl} to compute the transitive closure of equivalence classes as described in §4.1. In contrast, there are only 3 partial equivalence classes for the example, shown in Figure 3c since they depend only on local forwarding behavior.

Note that partial equivalence classes do not guarantee equivalent end-to-end behavior of packets, only local forwarding. For instance the packets $\langle 10.7.1.2,\ 255\rangle$ and $\langle 10.7.1.2,\ 1\rangle$ belong to the same partial equivalence class (2) in Figure 3c. Yet when sent from v_1, the latter packet will be dropped at v_2 while the former will be forwarded to v_3.

Of importance is that the definition of partial equivalence classes depends only on the rule transformations τ applied rather than the application of $\tau(\mathcal{H}_i)$ to some set of packets. This means that we can compute partial equivalence classes efficiently for header stacks using techniques similar to that of prior work [46] by looking only at the top of the stack.

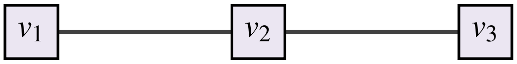
4.3 Verification algorithm overview

Given a set of (changed) partial equivalence classes and a property P, our objective is to check whether P holds for all packets in all of the (changed) partial equivalence classes.

Our approach is as follows: given a set of partial equivalence classes $\{\mathcal{H}_1,\dots,\mathcal{H}_n\}$ we start by exploring the reachable paths from every source node using a depth-first search. At each node u, packets in the partial equivalence class for $\mathcal{H}_i$ will all have the same next hop v and transformation τ (by definition). We proceed to apply $\tau(\mathcal{H}_i)$ to get some new set of packets $\mathcal{H}_i'$. Because $\mathcal{H}_i'$ may partially overlap with one or more existing partial equivalence classes, we identify all other partial equivalence classes $\mathcal{H}_{i_1},\dots,\mathcal{H}_{i_m}$ such that $\forall j \in \{1,..,m\}, \mathcal{H}_i' \cap \mathcal{H}_{i_j} \neq \emptyset$. We then continue the search with each subset $(\mathcal{H}_i' \cap \mathcal{H}_{i_j})$.

Representing header stacks. We still have the problem of symbolically representing the infinite space of stacks of headers $\mathcal{H}^*$. To do so, we use a decomposed representation where we model a set of header stacks as a concrete stack of symbolic header sets. For instance, suppose the set of reachable header stacks at a given node is $\{\varepsilon \cdot h_1 \cdot h_2,\ \ \varepsilon, \cdot h_3 \cdot h_4\}$. We instead represent this set as the stack of header sets given by the stack $\varepsilon \cdot \{h_1,h_3\} \cdot \{h_2,h_4\}$.

Of course, this decomposed representation naturally over approximates the set of headers (e.g., it would appear that $\varepsilon \cdot h_1 \cdot h_4$ is a reachable header stack). However, by carefully tracking the transformations that modify the stack (e.g., that



rule	p	e	m	τ
r_0	100	$\langle v_1, v_1\rangle$	$\phi_{\text{ttl}}(\{0\})$	τ_{drop}
r_1	200	$\langle v_1, v_2\rangle$	$\phi_{\text{dst}}(10.7.1.0/24)$	$\tau_{\text{push}} \circ \tau_{\text{ttl}}$
r_2	300	$\langle v_1, v_2\rangle$	$\mathcal{H}^*$	τ_{drop}
r_3	100	$\langle v_2, v_2\rangle$	$\phi_{\text{ttl}}(\{0\})$	τ_{drop}
r_4	200	$\langle v_2, v_3\rangle$	$\phi_{\text{dst}}(10.7.1.0/24)$	$\tau_{\text{pop}} \circ \tau_{\text{ttl}}$
r_5	300	$\langle v_2, v_2\rangle$	$\mathcal{H}^*$	τ_{drop}
r_6	100	$\langle v_3, v_3\rangle$	$\phi_{\text{ttl}}(\{0\})$	τ_{drop}
r_7	200	$\langle v_3, v_3\rangle$	$\mathcal{H}^*$	τ_{delv}

(a) Example network topology and rules.

(1) $\phi_{\text{dst}}(10.7.1.0/24) \cap \phi_{\text{ttl}}(\{0\})$
(2) $\phi_{\text{dst}}(10.7.1.0/24) \cap \phi_{\text{ttl}}(\{1\})$
... ...
(256) $\phi_{\text{dst}}(10.7.1.0/24) \cap \phi_{\text{ttl}}(\{255\})$
(257) $\mathcal{H} - \phi_{\text{dst}}(10.7.1.0/24)$

(b) Equivalence classes computed by APKeep [46].

(1) $\phi_{\text{ttl}}(\{0\})$
(2) $\phi_{\text{dst}}(10.7.1.0/24) \cap \phi_{\text{ttl}}(\{1,..,255\})$
(3) $\mathcal{H} - \phi_{\text{dst}}(10.7.1.0/24) \cap \phi_{\text{ttl}}(\{1,..,255\})$

(c) Partial equivalence classes computed by KATRA.

Figure 3: Running example of computing reachability in a simple multilayer network. Example network has headers consisting of a destination IP and a time-to-live (TTL) field: $h \in \mathcal{H} = \langle d, t\rangle$ where $d \in \{0,..,2^{32}-1\}$ and $t \in \{0,..,255\}$. The transformation τ_{ttl} decrements the TTL field. (a) shows APKeep equivalence classes for the single-layer version of the network and (b) Katra's partial equivalence classes for the multi-layer version.

only h_1 leads to h_2 and only h_3 leads to h_4), this representation remains precise. On the other hand, the decomposed representation is convenient because it allows for modeling arbitrary sized stacks and can execute τ_{push} and τ_{pop} cheaply on the symbolic representation since it is just a concrete stack operation. It also lets us leverage existing efficient data structures such as those based on BDDs, to manipulate the stacks despite not having a fixed size.

Given a decomposed stack of header sets $\sigma = \varepsilon \cdot H_1 \cdot \ldots \cdot H_n$ the usual definitions for τ_{push} and τ_{pop} apply, and we use a definition of a transformation τ applied to stacks: $\tau(\sigma \cdot h) = \sigma \cdot \tau(H)$. One drawback with this definition is that the headers at different layers of the stack lose dependencies between them. For instance, if the stack $\varepsilon \cdot H_1 \cdot H_1$ is filtered and becomes $\varepsilon \cdot H_1 \cdot H_2$, it may be that the new stack should be $\varepsilon \cdot H_2 \cdot H_2$ since only those packets with the inner header in H_2 would have pushed to headers that later survived the filter. In general, we track the transformations applied to the headers at each layer of the stack, and then "repair" the stack on demand whenever our representation is at risk of losing precision.

4.4 Layered verification algorithm

The algorithm for verifying arbitrarily layered networks is shown in Algorithm 1. CheckProperty takes as input the network $\mathcal{N}$, the partial equivalence class (e.g., one that changed after a rule insertion or deletion) $\mathcal{H}_i$, a set of source nodes S, a destination node d, and a path property P to check for each pair of source and destination.

The algorithm starts by running a depth-first search from each source node $s \in S$ (line 7) and tracking the visited nodes. Each node in the algorithm contains (i) a topology node, (ii) the current partial equivalence class (initially $\mathcal{H}_i$), and (iii) the current symbolic stack (initially $\varepsilon \cdot \mathcal{H}_i$). An invariant of the algorithm is that the top of the symbolic stack is a subset of the current partial equivalence class.

The depth-first search first looks up the next hop (edge and transformation τ on line 11) for the current partial equivalence class $\mathcal{H}_i$ and node u. It then applies the transformation τ to the current stack (line 12). If τ is undefined for this stack, then the trace is terminated and the algorithm checks the property P on the path (stored in the previous pointers starting at u on line 14). If the property fails, it returns a counter example.

Otherwise, the algorithm inspects the new top of the stack σ and finds all new overlapping partial equivalence classes $\mathcal{H}_{i_j}$ (line 16). For each, it computes a new stack σ' (line 17) obtained by restricting the top of the stack to this new partial equivalence class. If the top of the stack changed it then "repairs" the rest of the stack (line 20). We go into this operation in more detail in §4.5. Afterwards, the algorithm creates a new node for the next hop v with the new partial equivalence class $\mathcal{H}_{i_j}$ and the new stack σ' (line 20).

At this point the algorithm marks u as visited (line 22) checks if the new node creates an infinite loop (line 25). The details of this check are complex and are covered in detail in §5.1. Finally, if the new node v has not yet been visited, it recursively calls Dfs from this new node (line 28).

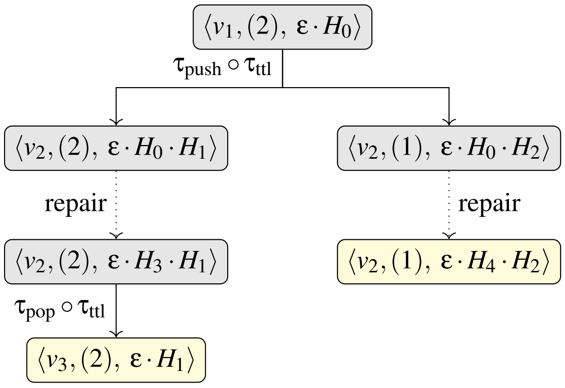
Example. We can see an application of Algorithm 1 in Figure 4. This shows the DFS trace produced for the earlier example network shown in Figure 3 that uses a time-to-live field. The execution is shown for the partial equivalence class $\mathcal{H}_i = (2)$, which corresponds to packets in the set $H_0 = \phi_{\text{dst}}(10.7.1.0/24) \cap \phi_{\text{ttl}}(\{1,..,255\})$. Initially the algorithm starts in partial equivalence class (2) with the stack $\varepsilon \cdot H_0$. From here, the algorithm discovers that the next hop is v_2 and the transformation is $\tau_{\text{push}} \circ \tau_{\text{ttl}}$. The result of applying this transformation to $\varepsilon \cdot H_0$ is two new sets of stacks corre-

Algorithm 1: Reachability for layered networks.

```
   Input: Network $\mathcal{N}$, partial equivalence class $\mathcal{H}_i$,
          Source locations $S$, Property $P$
   Output: Counterexample, or null if none
 1 Procedure CheckProperty($\mathcal{N}$, $\mathcal{H}_i$, $S$, $P$)
 2     visited ← $\emptyset$
 3     for s in $S$ do
 4         $u$ ← new Node(s, $\mathcal{H}_i$, $\varepsilon \cdot \mathcal{H}_i$)
 5         if $u \notin$ visited then
 6             $u$.previous ← null
 7             trace ← Dfs($\mathcal{N}$, $P$, visited, $u$, 0)
 8             if trace ≠ null return trace
 9     return null
10 Procedure Dfs($\mathcal{N}$, $P$, visited, $u$, $i$)
11     $\langle \text{edge}, \tau \rangle$ ← Forward($\mathcal{N}$, $u$.loc, $u$.ec)
12     $\sigma \leftarrow \tau(u.\text{stack})$
13     if $\sigma$ undefined then
14         return (if $P(u)$ then null else GetTrace($u$))
15     nexthops ← $\emptyset$
16     for $\mathcal{H}_{i_j}$ in OverlappingEcs(top($\sigma$)) do
17         $\sigma' \leftarrow \text{bot}(\sigma) \cdot (\text{top}(\sigma) \cap \mathcal{H}_{i_j})$
18         if top($\sigma$) ≠ top($\sigma'$) or $|\sigma| \neq |\sigma'|$ then
19             $\sigma'$ ← Repair($\sigma'$)
20         $v$ ← new Node(tgt(edge), $\mathcal{H}_{i_j}$, $\sigma'$)
21         nexthops ← nexthops ∪ $\{\langle \tau, v \rangle\}$
22     visited ← visited ∪ $\{u\}$
23     for $\langle \tau, v \rangle$ in nexthops do
24         $v$.previous ← $\langle \tau, u \rangle$
25         if HasLoop($v$, visited) then
26             return GetTrace($v$)
27         if $v \notin$ visited then
28             trace ← Dfs($\mathcal{N}$, $P$, visited, $v$, $i+1$)
29             if trace ≠ null return trace
30     return null
```

Algorithm 2: Unbounded loop check.

```
Input: Graph node $u$, and visited nodes visited
Output: Boolean for if there is an infinite/finite loop.
Procedure HasLoop($u$, $visited$)
    $\mathcal{C} \leftarrow \{n \mid n \in \text{visited}, n.\text{loc} = u.\text{loc}\}$
    $c \leftarrow u$.previous
    $\mu \leftarrow |u.\text{stack}|$
    while $c \neq$ null and $\mathcal{C} \neq \emptyset$ do
        $\mu \leftarrow \text{Min}(\mu, |c.\text{stack}|)$
        if $c \in \mathcal{C}$ then
            $\mathcal{C} \leftarrow \mathcal{C} - \{c\}$
            $\gamma$ ← LCS($u$.stack, $c$.stack)
            if $\mu > |c.stack| - \gamma$ then
                return true
        $c \leftarrow c$.previous
    return false
```


$\langle v_1, (2),\ \varepsilon \cdot H_0 \rangle$
$\tau_{\text{push}} \circ \tau_{\text{ttl}}$
$\langle v_2, (2),\ \varepsilon \cdot H_0 \cdot H_1 \rangle$
$\langle v_2, (1),\ \varepsilon \cdot H_0 \cdot H_2 \rangle$
repair
repair
$\langle v_2, (2),\ \varepsilon \cdot H_3 \cdot H_1 \rangle$
$\langle v_2, (1),\ \varepsilon \cdot H_4 \cdot H_2 \rangle$
$\tau_{\text{pop}} \circ \tau_{\text{ttl}}$
$\langle v_3, (2),\ \varepsilon \cdot H_1 \rangle$

$$\begin{aligned}
H_0 &= \phi_{\text{dst}}(10.7.1.0/24) &\cap\ & \phi_{\text{ttl}}(\{1,..,255\}) \\
H_1 &= \phi_{\text{dst}}(10.7.1.0/24) &\cap\ & \phi_{\text{ttl}}(\{1,..,254\}) \\
H_2 &= \phi_{\text{dst}}(10.7.1.0/24) &\cap\ & \phi_{\text{ttl}}(\{0\}) \\
H_3 &= \phi_{\text{dst}}(10.7.1.0/24) &\cap\ & \phi_{\text{ttl}}(\{2,..,255\}) \\
H_4 &= \phi_{\text{dst}}(10.7.1.0/24) &\cap\ & \phi_{\text{ttl}}(\{1\})
\end{aligned}$$

Figure 4: Example execution of Algorithm 1 for the partial equivalence class $\mathcal{H}_i = (2)$ from the example in Figure 3.

sponding to different partial equivalence classes. The first is $\varepsilon \cdot H_0 \cdot H_1$, which remains in partial equivalence class (2). The second is $\varepsilon \cdot H_0 \cdot H_2$, which now falls into partial equivalence class (1) since the TTL field reaches zero. In both cases, we "repair" the stack since the first header may be wrong. The results are given by $\varepsilon \cdot H_3 \cdot H_1$ and $\varepsilon \cdot H_4 \cdot H_2$. Those packets in partial equivalence class (1) are now dropped since the TTL field is 0. And the remaining packets are forwarded to v_3, decapsulated, and eventually delivered.

4.5 Repairing the stack

Recall in the example in Figure 4, the initial state is $\langle v_1, (2), H_0 \rangle$ capturing all packets for the destination prefix with TTL greater than zero. After being transformed by $\tau_{\text{push}} \circ \tau_{\text{ttl}}$, the resulting headers for partial equivalence class (2) are given by the stack $\varepsilon \cdot H_0 \cdot H_1$. Regrettably, H_0 is no longer correct because it contains a packet with a TTL field of 1, which would be 0 after the TTL decrement and thus no longer be part of H_1, which has TTL values in $\{1,..,254\}$.

The problem generally is that after restricting the top of the stack (Algorithm 1, line 17), the bottom of the stack may contain too many headers. To repair the stack, we reverse all transformations applied to the current stack to recover the initial set of packets from the source that will eventually lead to the new restricted stack. We then replay the transformations forward with the correct initial set to simulate the construction of the repaired stack as though we had started with the set that takes into account the later restriction.

Definition 4.4 (Transformation Inverse). *Given a transformation τ for sets of stacks, we define its inverse as $\tau^{-1}(H^*) = \{\sigma \in \mathcal{H}^* \mid \tau(\sigma) \in H^*\}$.*

Assume we have a sequence of stacks and transformations starting from the initial state of the depth-first search: $\sigma_1 \xrightarrow{\tau_1} \sigma_2 \xrightarrow{\tau_2} \dots \sigma_{n-1} \xrightarrow{\tau_{n-1}} \sigma_n$. We compute:

$$\begin{array}{lcl} \sigma_{\text{init}} & = & (\tau_1^{-1} \circ \dots \circ \tau_n^{-1})(\sigma_n) \\ \sigma_{\text{repair}} & = & (\tau_n \circ \dots \circ \tau_1)(\sigma_{\text{init}}) \end{array}$$

Example. We clarify this idea through an example. In the DFS shown in Figure 4, at node $\langle v_2, (2), H_0 \cdot H_1 \rangle$ we perform a stack repair operation. To do so, we compute σ_{init}:

$$\begin{array}{lll} & \sigma_{\text{init}} & \textit{compute } \sigma_{\text{init}} \\ = & (\tau_{\text{push}} \circ \tau_{\text{ttl}})^{-1}(H_0 \cdot H_1) & \textit{unfold definition} \\ = & \tau_{\text{push}}^{-1}(\tau_{\text{ttl}}^{-1}(H_0 \cdot H_1)) & \textit{function composition} \\ = & \tau_{\text{push}}^{-1}(H_0 \cdot H_3) & \textit{inverse of } \tau_{\text{ttl}} \\ = & H_3 & \textit{inverse of } \tau_{\text{push}} \end{array}$$

$$\begin{array}{lll} & \sigma_{\text{repair}} & \textit{compute } \sigma_{\text{repair}} \\ = & (\tau_{\text{push}} \circ \tau_{\text{ttl}})(H_3) & \textit{unfold definition} \\ = & \tau_{\text{ttl}}(\tau_{\text{push}}(H_3)) & \textit{function composition} \\ = & \tau_{\text{ttl}}(H_3 \cdot H_3) & \textit{definition of } \tau_{\text{push}} \\ = & H_3 \cdot H_1 & \textit{definition of } \tau_{\text{ttl}} \end{array}$$

This result is given by node $\langle v_2, (2), H_3 \cdot H_1 \rangle$ in Figure 4.

4.6 Property expressiveness

For efficiency, our algorithm concerns itself primarily with checking path properties P that are "subpath closed":

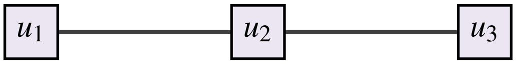
Definition 4.5 (Subpath Closed). *A property P is subpath closed if whenever P holds on a sequence of nodes $u_0, \dots, u_n$, it also holds on any subsequence $u_j, u_{j+1}, \dots, u_n$ for $j \geq 0$.*

Subpath-closed properties include reachability to a destination, loop-freedom, and network isolation. We focus on this subset of properties because they permit an efficient implementation by avoiding exploring previously visited nodes (Algorithm 1, line 5). However, this is not an inherent limitation of our algorithm – with only minor changes it can be used to check any path properties for packets, albeit at greater cost since we can not reuse previously visited nodes.

5 Algorithm Correctness

We now prove that Algorithm 1 is sound with respect to our concrete packet semantics from §3.3. But first we must define what it means for a DFS state to contain a located packet:

Definition 5.1 (DFS overapproximation). *For a located packet ℓ and Dfs node u, we write $\ell \in u$ if $\ell = \langle v, \sigma \rangle$ and u.loc $= v$ and $\sigma \in u$.stack and $\text{top}(\sigma) \in u$.ec.*



rule	p	e	m	τ
r_0	100	$\langle u_1, u_1 \rangle$	$\phi_{\text{dst}}(10.0.1.0/24)$	$\tau_{\text{push}} \circ \tau_d \circ \tau_{\text{push}}$
r_1	100	$\langle u_2, u_2 \rangle$	$\phi_{\text{dst}}(d)$	τ_{pop}
r_2	200	$\langle u_2, u_3 \rangle$	$\phi_{\text{dst}}(10.0.1.0/24)$	τ_{push}
r_3	300	$\langle u_2, u_3 \rangle$	$\mathcal{H}^*$	τ_{drop}
r_3	100	$\langle u_3, u_2 \rangle$	$\mathcal{H}^*$	τ_{push}

Figure 5: Example network with an infinite loop for the 10.0.1.1 address. The value d can be any other IP address.

Now given this definition, we state the soundness of Algorithm 1 as follows by relating the concrete semantics to the DFS calls made in the algorithm. For the simplicity of the proof, we elide the visited set optimization (lines 5 and 27) and the loop check (line 25). We revisit the loop check for termination in §5.1

Theorem 5.1 (Soundness). *For any network $\mathcal{N}$, partial equivalence class $\mathcal{H}_j$, node v, header $h \in \mathcal{H}_j$, located packet $\ell = \langle v, \varepsilon \cdot h \rangle$, and step $i \geq 0$, if not $\mathcal{N} \otimes \langle i, \ell \rangle$ then after calling* CheckProperty$(\mathcal{N}, \mathcal{H}_j, \{v\}, P)$ *there will eventually be a call to* Dfs$(\mathcal{N}, P, _, u, i)$ *for some node u such that* $\text{top}([\![\mathcal{N}]\!]_i \ell) \in u$.

Proofs are included as extra material in the appendix.

Corollary 5.1 (Property checking). *If i is the smallest step such that $\mathcal{N} \otimes \langle i, \ell \rangle$ then Algorithm 1 checks $P(u)$ for some DFS node u such that* $\text{top}([\![\mathcal{N}]\!]_i \ell) \in u$.

5.1 Infinite Loops and Termination

While Theorem 5.1 says that Algorithm 1 is sound, it says nothing about whether it will terminate. Intuitively, a network $\mathcal{N}$ contains a loop for a header h, whenever that packet will visit a node infinitely often in the future[2]. Catching infinite loops is vital since otherwise Algorithm 1 may not terminate. Finding loops in layered networks is surprisingly challenging since the space of header stacks is infinite and no stack need repeat to have an infinite loop.

We start by defining a loop for a given header:

Definition 5.2 (Network Loop). *Given a network $\mathcal{N}$, an input located packet ℓ induces a loop if there exists a step index $i \in \mathbb{N}$ for the start of the loop such that for all steps $j \in \mathbb{N}$ where $j \geq i$, there exists a future $k \in \mathbb{N}$ such that:*

$$\begin{array}{ll} (1) & |[\![\mathcal{N}]\!]_j \ell| < |[\![\mathcal{N}]\!]_k \ell| \\ (2) & \text{top}(\text{hops}([\![\mathcal{N}]\!]_j \ell)) = \text{top}(\text{hops}([\![\mathcal{N}]\!]_k \ell)) \end{array}$$

[2] Note: networks modeling the TTL field like in Figure 7 do not have a loop in the algorithmic sense because the packet will eventually expire after a finite number of steps. Such issues can be caught with an appropriate property P that looks for packets that eventually expire with TTL 0.

In other words, a loop exists if beyond some point in the trace (i) the trace will continue to grow forever and repeatedly visit the same nodes in the network.

Example. We demonstrate the difficulty of detecting infinite loops in Figure 5. Unlike in single layer networks, loops in multi-layer networks may be transient even when the top of the stack repeats at the same node because of implicit state lower in the header stack. Further, any given stack may not repeat even when an infinite loop exists since the stack can keep growing. Consider a trace for the example in Figure 5 for traffic sent from u_1 with the 10.0.1.1 destination.

$\langle u_1, 10.0.1.1\rangle$	*encapsulate twice*
$\langle u_2, 10.0.1.1 \cdot d \cdot d\rangle$	*pop*
$\langle u_2, 10.0.1.1 \cdot d\rangle$	*pop*
$\langle u_2, 10.0.1.1\rangle$	*forward to u_3*
$\langle u_3, 10.0.1.1 \cdot 10.0.1.1\rangle$	*forward to u_2*
$\langle u_2, 10.0.1.1 \cdot 10.0.1.1 \cdot 10.0.1.1\rangle$	*forward to u_3*
$\ldots$	

Note that the top of stack d is repeated at node u_2, however, this is not the cause of the infinite loop since eventually this outer header is removed and the forwarding proceeds according to the inner header for the 10.0.1.1 address. Later, however, there is an infinite loop despite the stack never repeating exactly at any node in the trace.

Necessary and sufficient conditions. Suppose we have a current header stack $\sigma \cdot h$ at node u, and later on we arrive at u once more, but with header stack $\sigma \cdot \sigma' \cdot h$ with the same shared prefix σ. Moreover, assume that between visiting u twice, the rules never examine the contents of σ. If these conditions hold then the top of the stack h "regenerates" itself without needing context from σ. In this case, we can infer that there will be an infinite loop at u given by: $\langle u, \sigma \cdot h\rangle \longrightarrow \langle u, \sigma \cdot \sigma' \cdot h\rangle \longrightarrow \langle u, \sigma \cdot \sigma' \cdot \sigma' \cdot h\rangle \longrightarrow \ldots$ This idea is similar to repeating heads from the verification of pushdown systems [36] and we prove that this condition is both sufficient and necessary for a permanent loop:

Theorem 5.2 (Loop conditions). *Given a network $\mathcal{N}$ over $\mathcal{H}^*$, an input ℓ induces a loop if and only if there exists $i, k \in \mathbb{N}$, $\sigma, \sigma' \in \mathcal{H}^*$, and $h \in \mathcal{H}$ such that:*

(1) $\mathrm{top}([\![\mathcal{N}]\!]_i \ell) = \langle u, \sigma \cdot h\rangle$

(2) $\mathrm{top}([\![\mathcal{N}]\!]_k \ell) = \langle u, \sigma \cdot \sigma' \cdot h\rangle$

(3) $\forall j,\ i < j < k \Rightarrow \exists v, \sigma'',\ \mathrm{top}([\![\mathcal{N}]\!]_j \ell) = \langle v, \sigma \cdot \sigma''\rangle$

Loop detection algorithm. Based on the insights from Theorem 5.2, we develop an efficient procedure for checking loops during traversal, which is described in Algorithm 2. Given the current node (u) in the DFS, and the visited nodes (visited) the procedure checks for a loop by looking up all candidate nodes ($\mathcal{C}$) for the same current topology location (u.loc, line 2). The algorithm walks backwards through the current path (line 5) and computes the longest common suffix (LCS) γ between the tops of the stacks for the two nodes u and c (line 9) while also tracking the minimum stack size μ between the two nodes. If μ is greater than $|c.\text{stack}| - \gamma$ (where γ generalizes h in the loop conditions) then there is a loop (line 11). If the set of candidate loop nodes has been exhausted, the algorithm terminates early.

```
1  // instantiate a new network verifier
2  var headerType = new HeaderType(
3      ("dstip", 32), ("srcip", 32));
4  var nv = new NetworkVerifier(headerType);
5
6  // build the network topology
7  var (n1, n2) = nv.GetOrAddNodes("n1", "n2");
8  var (e12, e21) = nv.GetOrAddBiEdge(n1, n2);
9
10 // register the properties we want to monitor
11 nv.AddCheck(new LoopCheck(nv.AllHeaders()));
12
13 // create new prioritized forwarding rules
14 var r = nv.CreateRange(
15     (10, 20), (0, uint.MaxValue));
16 var t = nv.Seq(nv.Push(), nv.Set("dstip", 10));
17 var rule1 = new Rule(100, e12, r, t);
18 var rule2 = new Rule(100, e21, r, nv.Pop());
19
20 // find violations from adding rules.
21 var violations1 = nv.AddRule(rule1);
22 var violations2 = nv.AddRule(rule2);
23 Assert.AreEqual(1, violations2.Count);
```

Figure 6: Example use of the KATRA verification API.

6 Implementation

We have built an incremental verification system, KATRA for layered networks based on the idea presented. KATRA is implemented as a C# library and is written in around 8K lines of code. KATRA's implementation for computing header equivalence classes is based on the algorithm from [46], but is modified to incrementally compute the *minimal* set of partial equivalence classes (see §4.2). An example of an API for the tool is shown in Figure 6. The tool is programmable and is parameterized by the format of the header (e.g., MPLS vs. IPv4) that the user wants to check (line 1)[3]. Our implementation extends §3.2 to support mulipath routing.

Optimizations. KATRA makes use of several optimizations to scale. One key challenge is that the use of partial equivalence classes (§4.2) means that we must find overlapping equivalence classes during traversal (Algorithm 1, line 16). To make this operation fast, for every packet set H we keep a pair of $\langle b, H\rangle$ where H is the set itself modeled as a BDD [10], and b is a multi-dimensional bounding box that overapproximates

[3] To model different headers in different layers (e.g., Ethernet and IPv4 headers), one can define a "master" header with the union of fields across headers along with a field indicating which header is currently being used.

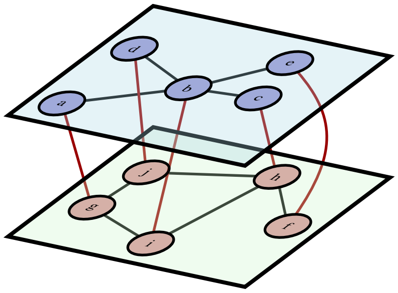


Figure 7: Example of a layered benchmark network with layers $\ell = 2$ and nodes $n = 5$.

the set of headers in H. When sets are unioned or intersected, the corresponding bounding boxes are grown or shrunk to remain safe overapproximations for the sets of headers.

Keeping bounding boxes for header sets allows for the use of fast collision detection data structures. We leverage bounding volume hierarchies [40], which are hierarchical balanced trees of bounding volumes used in game engines to quickly eliminate possible collisions.

Our implementation also examines header sets H and determines if the fields lie on prefix boundaries (e.g., for IPv4 prefix-based forwarding). If so, it uses an optimized trie data structure to accelerate the collision detection.

7 Evaluation

We are primarily interested in evaluating the performance of KATRA relative to a straightforward extension of prior work that models packets with a fixed (bounded) number of headers N. Of course, this approach requires a user to specify N and may be unsound when N is not large enough to handle the maximum stack possible in the network. However, if N is chosen carefully this provides a reasonable comparison point.

7.1 Different implementations

To compare the approach in KATRA with that of duplicate headers (DUP), we instantiate our framework (§6) with two types of headers. For DUP, we instantiate the verifier with a header that is similar to that of Figure 6 (line 2) but extended to a full IPv4 header, and replicated N times. We choose N to account for the maximum amount of layering in each benchmark and do not evaluate DUP on networks that contain unbounded loops, since it will give incorrect results.

Each field in the DUP header has versions f_1 to f_N and f_1 represents the outermost header (top of stack). The push operation is implemented by copying each field f_i to f_{i+1}, its next layer version, and the bottom header is lost in the process if the stack exceeds size N. The pop operation is implemented similarly by copying each field f_{i+1} to f_i.

Single layer performance. APKeep was demonstrated to outperform prior incremental verifiers while also being more robust to multi-dimensional rules [46]. However, since APKeep is not open source, we instead use our implementation of KATRA, which uses a similar base algorithm to compare results. We ran KATRA on the same datasets reported on in the APKeep paper and originally released by Deltanet [20]. We found the performance for these single layer networks to be similar to the times reported on by APKeep, and as such do not report on the results here. Since the implementation performance is comparable, going forward we report only the times from different instantiations of KATRA.

Moreover, instantiating DUP in KATRA allows us to directly compare our algorithm to a naive solution without other factors coming into play. For example, DUP also makes use of our partial equivalence class reduction, our fast collision detection data structure, and other optimizations.

7.2 Performance on multilayer networks

To measure the performance of KATRA for multilayer networks (i.e., with stack size greater than 1), we generated a parameterized set of benchmark networks.

Benchmark description. The benchmarks have two parameters: the number of layers ℓ in the network, and the number of nodes per layer n. The first layer represents the physical network, while each layer $i > 1$ represents an overlay network built on top of layer $i-1$. Each link in the layer i in the network is implemented by encapsulating a packet and forwarding it according to the destination prefix for the tunnel endpoint in layer $i-1$. Routing in each layer is configured to announce and propagate routes along shortest paths. For each $\langle \ell, n \rangle$ pair, we generate the topologies as random connected graphs and map nodes in each overlay to nodes in the underlay for the purpose of establishing tunnel endpoints.

An example of such a network with $\ell = 2$ and $n = 5$ is shown in Figure 7. In the example, to forward traffic between layer 2 nodes b and e, traffic is encapsulated and forwarded from i to f via h in layer 1. For such networks, there are a total of $O(\ell \cdot n^2)$ forwarding rules.

The first property we check is reachability between all source and destination nodes in the outermost layer ℓ for all advertised subnets. This strategy forces KATRA to reason about the forwarding behavior at every single layer. Because these reachability properties are violated while tunnels are being established at different layers, for this benchmark we disable property checking while connectivity is not expected.

***Performance of* KATRA *compared to* DUP.** We show the total verification time of KATRA and DUP in Figure 8 and Figure 9. Figure 8 shows the total time spent recomputing partial equivalence classes for both approaches. KATRA is faster than DUP because DUP must represent significantly larger headers in order to capture the full stack. This leads to larger packet set representations in the BDD library and more expensive set and transform operations.

Similarly, Figure 9 shows the total time spent checking

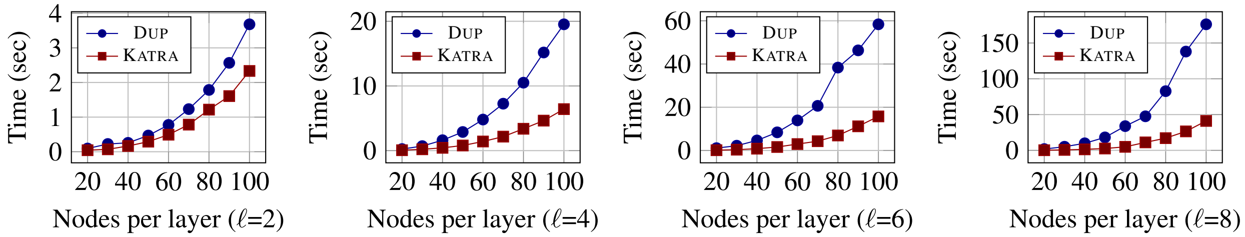


Figure 8: Total time spent recomputing partial equivalence classes for an approach based on duplicate headers Dup vs. Katra.

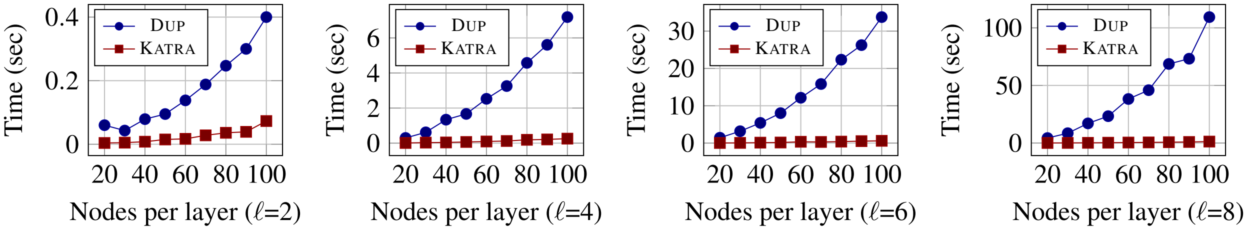
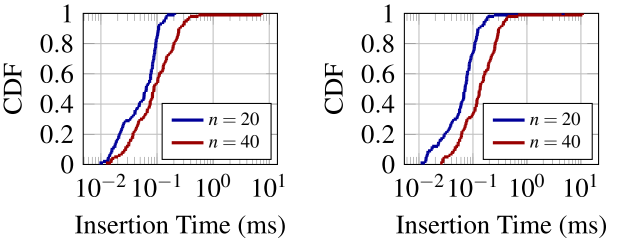


Figure 9: Total time spent checking end-to-end reachability for an approach based on duplicate headers Dup vs. Katra.

reachability for Katra and Dup. There is a similar trend, with Katra's decomposed stack set representation leading to a large speedup over that of Dup. In both cases, the speedup of Katra grows with both the number of layers ℓ and the number of nodes per layer n in the graph. For instance, at $\ell = 6$ and $n = 100$, property checking is nearly 50x faster.

Katra *rule update time.* The total verification time grows quickly in part because the number of rules needed to implement the network design is proportional to the square of the size of the network. However, looking at the time for each individual rule update in Figure 10, essentially all updates execute in under 1ms. The graphs show the CDF for rule insertion time in milliseconds. In particular, the insertion time is relatively independent of ℓ yet increases slightly with n.



(a) Layers ℓ=2

(b) Layers ℓ=4

Figure 10: CDF of rule insertion time for (a) ℓ=2 and (b) ℓ=4. Both show results for nodes per layer $n = 20$ and $n = 40$.

7.3 Performance of loop checking

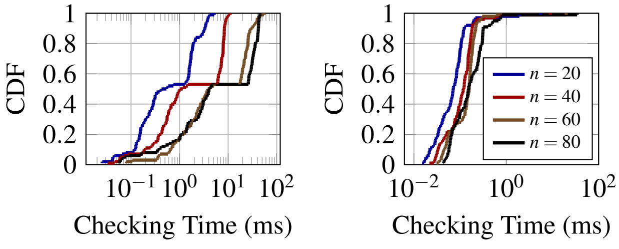
To evaluate the performance of Algorithm 2, we used example networks with $\ell = 2$ and replaced the reachability checks for each destination subnet with a single loop check for all packets. Unlike with reachability, this property gets rechecked after every single rule insertion. Since each link in layer 2 crosses many of the same previous nodes in layer 1, this forces Algorithm 2 to check for potential loops frequently.

A CDF of the rule insertion and loop checking time for each update are shown in Figure 11. We vary n from 20 to 80 in increments of 20 and compare the results. Figure 11a shows the checking time when rules are inserted in an arbitrary order. The time grows with the size of the network and can become high at the tail (e.g., around 40ms).

The reason why is that if a rule r with transformation τ_{pop} is inserted early, then every other rule insertion will affect the partial equivalence class for r. In other words, after a decapsulation a packet may now be in any partial equivalence class, so when any other partial equivalence class changes, the partial equivalence class for r must also be rechecked. At the extreme, this means that every rule insertion can require rechecking the entire network from scratch. This is inherent in the problem and is not unique to Katra (e.g., APKeep suffers a similar blowup for these networks).

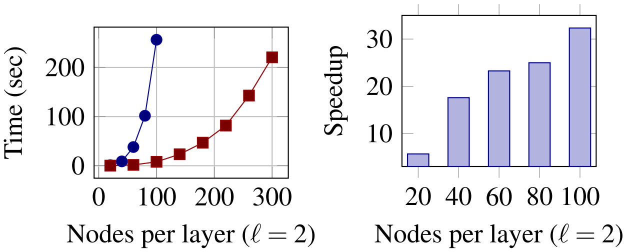
However, by slightly reordering rule updates, we can improve the performance significantly. In Figure 11b, we show the same results but where the rule insertion order is done in a way to delay the insertion of decapsulation rules. From the figure, we can see that in the latter case, the checking time remains well below 1ms for nearly all rules.

Since this benchmark requires checking the loop property after *all* rule updates, the performance improvement of Katra grows substantially over that of Dup. Figure 12 shows the total time to verify the loop-free property for all rule updates. It shows the performance for $\ell = 2$ layers where we cap the total verification time at 4 minutes. Dup times out after $n = 100$ with 20K rules while Katra can verify networks up to $n = 300$ with 180K rules. The relative speedup of Katra over Dup for $n = 20$ to $n = 100$ is shown in Figure 12b.



(a) Random order (b) Delayed decap

Figure 11: CDF of rule insertion time for ℓ=2 and $n = 20$ to $n = 80$ in increments of 20. Total checking time for loops is low when decapsulation rule insertion is delayed.



(a) Total time (b) Relative speedup

Figure 12: (a) Total verification time for DUP (blue) vs. KATRA (red) for $\ell = 2$ when checking for forwarding loops on every change. (b) Speedup ranges from 5x to 32x.

8 Related Work

KATRA is related to several threads of prior work:

Data plane verification. There is a long line of work on data plane verification, starting from the seminal work of Xie et. al. [27, 41, 42], and incremental verification starting with Veriflow [20, 26, 29, 46]. Most work on data plane verification has assumed stateless and transformation-free forwarding, with the exception of AP [42] and APKeep [46], which handle transformations (see §4). However, none of these works consider *layered* networks where encapsulation and decapsulation are pervasive. AP and APKeep can model finite header stacks (e.g., DUP from §7) but this approach can be unsound and can have poor performance, particularly when encapsulation is common. KATRA builds on prior work to enable incremental verification with transformations and layering.

Layered network verification. There has been little work on verifying multilayer networks. One related work in this area is Tiramisu [2], which can verify some combinations of layer 2 and 3 control plane routing protocols (e.g., BGP, iBGP, OSPF). However, Tiramisu is only superficially related to KATRA: (i) Tiramisu verifies control plane routing while KATRA verifies data plane forwarding, (ii) Tiramisu focuses on specific layering mechanisms (e.g., between iBGP and eBGP) while KATRA focuses on arbitrarily layered *data planes*, and (iii) KATRA is interested in *real-time* (millisecond) verification time for incremental changes.

Recent works on verifying MPLS label switching with fast failover [22–24] were the first to leverage the insight that label-based forwarding can be viewed as pushdown automata. The works use polynomial time algorithms to answer reachability questions for all possible failures using overapproximation. While they focus on reasoning about failures, we similarly leverage this insight that ideas from pushdown automata are useful for reasoning about stacks of headers. We generalize this reasoning from concrete label-based forwarding to symbolic forwarding (e.g., prefix-based forwarding) and also focus on *realtime* verification for changes.

There are significant differences in the actual algorithms. These prior works use saturation-based procedures to iteratively compute automata representations of (backward or forward) reachable configurations of the pushdown system. In contrast, our algorithm is an on-the-fly depth-first search over *symbolic* configurations, which include (partial) equivalence classes over the header space.

One work [24] considers abstractions based on network labels to reduce PDS size and proposes a CEGAR-style refinement procedure, which improves performance in many practical examples. Our symbolic configurations are also *abstractions* of the network state space, where the control state is a partial equivalence class in the header space located at a particular node in the network, and the stack is a word over these classes. These abstractions are refined lazily on-the-fly in our novel method for stack repair, such that any trace in our algorithm follows the specified network semantics.

Model checking of pushdown systems. More broadly, our work builds on prior work in model checking of pushdown systems [6, 9, 36], which can naturally represent sequential programs with recursive procedures. Similar to symbolic procedures for pushdown systems [12, 36], we also utilize BDDs [11] for efficient representation of the state space and use a notion similar to *repeating heads* [36] for detecting loops. However, rather than computing sets of reachable configurations, our procedure performs on-the-fly verification to soundly check reachability of located packets.

9 Conclusion

In this paper we have presented KATRA, the first real-time verifier for *layered* networks. KATRA extends incremental data-plane verification to the setting with unbounded header stacks. To do so, we introduced a new network model for layered networks and presented an efficient algorithm for such networks. The algorithm leverages a new idea of *partial* equivalence classes and keeps a decomposed symbolic stack representation that it lazily “repairs” as needed. Comparing KATRA against a solution based on header duplication, we showed that KATRA is 5x-32x faster for just 2 layers, and that its benefits grow with network size and layering.

References

[1] 3GPP. General Packet Radio Service (GPRS); GPRS Tunnelling Protocol (GTP) across the Gn and Gp interface, January 1999.

[2] 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, Santa Clara, CA, February 2020. USENIX Association.

[3] Amazon. Amazon ec2 secure and resizable compute capacity to support virtually any workload. https://aws.amazon.com/ec2/, 2021.

[4] Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. Netkat: Semantic foundations for networks. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 113–126, New York, NY, USA, 2014. ACM.

[5] John Backes, Sam Bayless, Byron Cook, Catherine Dodge, Andrew Gacek, Alan J. Hu, Temesghen Kahsai, Bill Kocik, Evgenii Kotelnikov, Jure Kukovec, Sean McLaughlin, Jason Reed, Neha Rungta, John Sizemore, Mark Stalzer, Preethi Srinivasan, Pavle Subotić, Carsten Varming, and Blake Whaley. Reachability analysis for aws-based networks. In Isil Dillig and Serdar Tasiran, editors, *Computer Aided Verification*, pages 231–241, Cham, 2019. Springer International Publishing.

[6] Thomas Ball and Sriram K. Rajamani. Bebop: a path-sensitive interprocedural dataflow engine. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering (PASTE)*, pages 97–103. ACM, 2001.

[7] Nikolaj Bjørner, Garvit Juniwal, Ratul Mahajan, Sanjit A. Seshia, and George Varghese. ddnf: An efficient data structure for header spaces. In *Haifa Verification Conference*, 2016.

[8] J. Border, M. Kojo, J. Griner, G. Montenegro, and Z. Shelby. Performance Enhancing Proxies Intended to Mitigate Link-Related Degradations. Internet Request for Comments, June 2001.

[9] Ahmed Bouajjani, Javier Esparza, and Oded Maler. Reachability analysis of pushdown automata: Application to model-checking. In *CONCUR '97: Concurrency Theory, Proceedings*, volume 1243 of *Lecture Notes in Computer Science*, pages 135–150. Springer, 1997.

[10] Karl S. Brace, Richard L. Rudell, and Randal E. Bryant. Efficient implementation of a bdd package. In *Proceedings of the 27th ACM/IEEE Design Automation Conference*, DAC '90, pages 40–45, New York, NY, USA, 1990. ACM.

[11] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.

[12] Javier Esparza and Stefan Schwoon. A BDD-based model checker for recursive programs. In *Computer Aided Verification, International Conference, CAV, Proceedings*, volume 2102 of *Lecture Notes in Computer Science*, pages 324–336. Springer, 2001.

[13] FlexiWAN. The world's first open source sd-wan & sase. https://flexiwan.com/, 2021.

[14] Klaus-Tycho Foerster, Yvonne-Anne Pignolet, Stefan Schmid, and Gilles Tredan. Local fast failover routing with low stretch. *SIGCOMM Comput. Commun. Rev.*, 48(1):35–41, apr 2018.

[15] B. Gleeson, A. Lin, J. Heinanen, Telia Finland, G. Armitage, and A. Malis. A Framework for IP Based Virtual Private Networks. Internet Request for Comments, February 2000.

[16] Google. Google cloud: Cloud computing services. https://cloud.google.com/, 2021.

[17] Google. Network intelligence center: Connectivity tests overview. https://cloud.google.com/network-intelligence-center/docs/connectivity-tests/concepts/overview, 2021.

[18] S. Hanks, Ltd. NetSmiths, T. Li, D. Farinacci, and P. Traina. Generic Routing Encapsulation (GRE). Internet Request for Comments, October 1994.

[19] Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. Achieving high utilization with software-driven wan. *SIGCOMM Comput. Commun. Rev.*, 43(4):15–26, August 2013.

[20] Alex Horn, Ali Kheradmand, and Mukul Prasad. Delta-net: Real-time network verification using atoms. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 735–749, Boston, MA, March 2017. USENIX Association.

[21] Karthick Jayaraman, Nikolaj Bjorner, 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, pages 200–213, New York, NY, USA, 2019. ACM.

[22] Jesper Stenbjerg Jensen, Troels Beck Krøgh, Jonas Sand Madsen, Stefan Schmid, Jiří Srba, and Marc Tom Thorgersen. P-rex: Fast verification of mpls networks with multiple link failures. In *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '18, page 217–227, New York, NY, USA, 2018. Association for Computing Machinery.

[23] Peter Gjøl Jensen, Dan Kristiansen, Stefan Schmid, Morten Konggaard Schou, Bernhard Clemens Schrenk, and Jiří Srba. Aalwines: A fast and quantitative what-if analysis tool for mpls networks. In *Proceedings of the 16th International Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '20, page 474–481, New York, NY, USA, 2020. Association for Computing Machinery.

[24] Peter Gjøl Jensen, Stefan Schmid, Morten Konggaard Schou, Jirí Srba, Juan Vanerio, and Ingo van Duijn. Faster pushdown reachability analysis with applications in network verification. In *Automated Technology for Verification and Analysis (ATVA), Proceedings*, volume 12971 of *Lecture Notes in Computer Science*, pages 170–186, 2021.

[25] Andrzej Kamisiński. Evolution of ip fast-reroute strategies. In *2018 10th International Workshop on Resilient Networks Design and Modeling (RNDM)*, pages 1–6, 2018.

[26] Peyman Kazemian, Michael Chang, Hongyi Zeng, George Varghese, Nick McKeown, and Scott Whyte. Real time network policy checking using header space analysis. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 99–111, Lombard, IL, April 2013. USENIX Association.

[27] Peyman Kazemian, George Varghese, and Nick McKeown. Header space analysis: Static checking for networks. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 113–126, San Jose, CA, April 2012. USENIX Association.

[28] S. Kent and K. Seo. Security Architecture for the Internet Protocol. Internet Request for Comments, August 2005.

[29] Ahmed Khurshid, Xuan Zou, Wenxuan Zhou, Matthew Caesar, and P. Brighten Godfrey. Veriflow: Verifying network-wide invariants in real time. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 15–27, Lombard, IL, 2013. USENIX.

[30] M. Mahalingam, D. Dutt, K. Duda, P. Agarwal, L. Kreeger, T. Sridhar, M. Bursell, and C. Wright. Virtual eXtensible Local Area Network (VXLAN): A Framework for Overlaying Virtualized Layer 2 Networks over Layer 3 Networks. Internet Request for Comments, August 2014.

[31] Microsoft. Microsoft azure: Cloud computing services. https://azure.microsoft.com/en-us/, 2021.

[32] J. Moy. Open Shortest Path First Protocol Version 2. Internet Request for Comments, April 1998.

[33] A. Atlas P. Pan, G. Swallow. Fast Reroute Extensions to RSVP-TE for LSP Tunnels. Internet Request for Comments, May 2005.

[34] C. Perkins. IP Encapsulation within IP. Internet Request for Comments, July 1996.

[35] E. Rosen, A. Viswanathan, and R. Callon. Multiprotocol Label Switching Architecture. Internet Request for Comments, January 2011.

[36] Stefan Schwoon. *Model checking pushdown systems*. PhD thesis, Technical University Munich, 2002.

[37] Oliver Spatscheck. Layers of success. *IEEE Internet Computing*, 17(1):3–6, 2013.

[38] 3GPP The Mobile Broadband Standard. 3gpp a global initiative. https://www.3gpp.org/, 2021.

[39] Bingchuan Tian, Xinyi Zhang, Ennan Zhai, Hongqiang Harry Liu, Qiaobo Ye, Chunsheng Wang, Xin Wu, Zhiming Ji, Yihong Sang, Ming Zhang, Da Yu, Chen Tian, Haitao Zheng, and Ben Y. Zhao. Safely and automatically updating in-network acl configurations with intent language. In *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM '19, page 214–226. Association for Computing Machinery, 2019.

[40] Ingo Wald. On fast construction of sah-based bounding volume hierarchies. In *2007 IEEE Symposium on Interactive Ray Tracing*, pages 33–40, 2007.

[41] G. G. Xie, Jibin Zhan, D. A. Maltz, Hui Zhang, A. Greenberg, G. Hjalmtysson, and J. Rexford. On static reachability analysis of ip networks. In *Proceedings IEEE 24th Annual Joint Conference of the IEEE Computer and Communications Societies.*, volume 3, pages 2170–2183 vol. 3, March 2005.

[42] Hongkun Yang and Simon S. Lam. Real-time verification of network properties using atomic predicates. *IEEE/ACM Trans. Netw.*, 24(2):887–900, April 2016.

[43] Kok-Kiong Yap, Murtaza Motiwala, Jeremy Rahe, Steve Padgett, Matthew Holliman, Gary Baldus, Marcus Hines, Taeeun Kim, Ashok Narayanan, Ankur Jain, Victor Lin, Colin Rice, Brian Rogan, Arjun Singh, Bert Tanaka, Manish Verma, Puneet Sood, Mukarram Tariq, Matt Tierney, Dzevad Trumic, Vytautas Valancius, Calvin Ying, Mahesh Kallahalla, Bikash Koley, and Amin Vahdat. Taking the edge off with espresso: Scale, reliability and programmability for global internet peering. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, page 432–445, New York, NY, USA, 2017. Association for Computing Machinery.

[44] Pamela Zave and Jennifer Rexford. The compositional architecture of the internet. *Commun. ACM*, 62(3):78–87, February 2019.

[45] Hongyi Zeng, Shidong Zhang, Fei Ye, Vimalkumar Jeyakumar, Mickey Ju, Junda Liu, Nick McKeown, and Amin Vahdat. Libra: Divide and conquer to verify forwarding tables in huge networks. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 87–99, Seattle, WA, April 2014. USENIX Association.

[46] Peng Zhang, Xu Liu, Hongkun Yang, Ning Kang, Zhengchang Gu, and Hao Li. Apkeep: Realtime verification for real networks. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 241–255, Santa Clara, CA, February 2020. USENIX Association.

Appendix

Theorem 5.1 (Soundness). *For any network $\mathcal{N}$, partial equivalence class $\mathcal{H}_j$, node v, header $h \in \mathcal{H}_j$, located packet $\ell = \langle v, \varepsilon \cdot h\rangle$, and step $i \geq 0$, if not $\mathcal{N} \otimes \langle i, \ell\rangle$ then after calling* CheckProperty$(\mathcal{N}, \mathcal{H}_j, \{v\}, P)$ *there will eventually be a call to* Dfs$(\mathcal{N}, P, _, u, i)$ *for some node u such that* top$(\llbracket \mathcal{N} \rrbracket_i \ell) \in u$.

Proof. The proof is by induction on the step i. For the sake of simplicity, we assume that lines 5 and 27 of Algorithm 1, which are optimizations using the visited set, are removed for the remainder of the proof.

Base case ($i = 0$) By assumption we have $\ell = \langle v, \varepsilon \cdot h\rangle$. From unfolding the definition of the semantics $\llbracket \mathcal{N} \rrbracket$ for the ($i = 0$) step, we obtain the following equality:

$$\text{top}(\llbracket \mathcal{N} \rrbracket_0 \ell) = \text{top}(\varepsilon \cdot \ell) = \ell = \langle v, \varepsilon \cdot h\rangle$$

Thus we must prove that there is a call to Dfs$(\mathcal{N}, P, _, u, 0)$ such that u.loc $= v$ and $\varepsilon \cdot h \in u$.stack and top$(\varepsilon \cdot h) \in u$.ec. This trivially follows from line 7 of Algorithm 1. Since $S = \{v\}$ (line 3), we see that $s = v$ (line 3) and therefore u.loc $= v$ as expected, and u.stack $= \varepsilon \cdot \mathcal{H}_j$ (line 4), which implies that $\varepsilon \cdot h \in u$.stack since $\varepsilon \cdot h \in \varepsilon \cdot \mathcal{H}_j \iff h \in \mathcal{H}_j$ by definition and this is an assumption. Finally, we have that top$(\varepsilon \cdot h) = h \in u$.ec or $h \in \mathcal{H}_j$ again by assumption.

Inductive case ($i > 0$) The proof proceeds by using the inductive hypothesis for step $i - 1$ to prove that the statement holds for step i. We list out our assumptions from the proof statement as well as the induction hypothesis below:

- not $\mathcal{N} \otimes \langle i, \ell\rangle$
- not $\mathcal{N} \otimes \langle i-1, \ell\rangle$
- top$(\llbracket \mathcal{N} \rrbracket_i \ell) = \langle v_1, \sigma_1\rangle$
- top$(\llbracket \mathcal{N} \rrbracket_{i-1} \ell) = \langle v_2, \sigma_2\rangle$
- there was a call to Dfs$(\mathcal{N}, P, _, u_2, i-1)$ for some u_2
- u_2.loc $= v_2$
- $\sigma_2 \in u_2$.stack
- top$(\sigma_2) \in u_2$.ec

Given these assumptions, we must prove that each of the following statements holds as a result:

- there is a call to Dfs$(\mathcal{N}, P, _, u_1, i)$ for some u_1
- u_1.loc $= v_1$
- $\sigma_1 \in u_1$.stack
- top$(\sigma_1) \in u_1$.ec

We walk through the lines of code in Algorithm 1 starting from the call to Dfs$(\mathcal{N}, P, _, u_2, i-1)$ that we know must have taken place. By our assumption that top$(\sigma_2) \in u_2$.ec, and from the definition of a partial equivalence class (same local forwarding for all packets in the equivalence class), we know the $\langle$edge$, \tau\rangle$ pair returned in line 11 must be equivalent to those of the semantics: $\tau =$ modify$(\Omega(v_2, \sigma_2))$ and edge $=$ edge$(\Omega(v_2, \sigma_2))$ from the semantic definition in §3.3. Evaluating $\llbracket \mathcal{N} \rrbracket_i \ell$ there are two cases:

Case 1: if $\tau(\sigma_2)$ is undefined, then we compute: $\llbracket \mathcal{N} \rrbracket_i \ell = \llbracket \mathcal{N} \rrbracket_{i-1} \ell$ and we observe that $\mathcal{N} \otimes \langle i-1, \ell\rangle$. In this case, the algorithm executes line 14 and terminates. Note that we do not call Dfs again, however, in this case the semantics were terminated at step $i - 1$ which contradicts the assumptions. Further, note that this is the minimal time step i at which $\mathcal{N} \otimes \langle i, \ell\rangle$ since we assumed not $\mathcal{N} \otimes \langle i-1, \ell\rangle$.

Case 2: if $\tau(\sigma_2)$ is defined, then we compute

$$\langle v_1, \sigma_1\rangle = \text{top}(\llbracket \mathcal{N} \rrbracket_i \ell) = \text{top}(\llbracket \mathcal{N} \rrbracket_{i-1} \ell \cdot \langle \text{tgt(edge)}, \tau(\sigma_2)\rangle) = \langle \text{tgt(edge)}, \tau(\sigma_2)\rangle$$

By the definition of τ lifted to sets, we know that because $\sigma_2 \in u$.stack then it follows that $\tau(\sigma_2) \in \tau(u$.stack$)$ (line 12) and therefore $\sigma_1 \in \tau(u$.stack$)$. The algorithm proceeds on line 16 to iterate over all partial equivalence classes that can intersect $\tau(u$.stack$)$. Because partial equivanence classes are disjoint and complete (see §4.2), there will be exactly one such $\mathcal{H}_{j_k}$ such that top$(\sigma_1) \in \mathcal{H}_{j_k}$. From this we can deduce line 17 will compute a new set of stacks σ' that must contain σ_1 – that is $\sigma_1 \in \sigma'$ by construction.

Line 19 of the algorithm updates σ' as Repair(σ'). Because $\sigma_1 \in \sigma'$ we must show that that $\sigma_1 \in$ Repair(σ') as well. To compute Repair(σ') we first compute $(\tau_1^{-1} \circ \ldots \circ \tau_n^{-1})(\sigma')$, which is equivalent to $\sigma_{\text{init}} = \{\sigma'' \mid (\tau_n \circ \ldots \circ \tau_1)(\sigma'') \in \sigma'\}$. Since σ_1 is the result of applying $(\tau_n \circ \ldots \circ \tau_1)$ to the initial header $\varepsilon \cdot h$, it follows that $\varepsilon \cdot h \in \sigma_{\text{init}}$. Because $\text{Repair}(\sigma') = \sigma_{\text{repair}} = (\tau_n \circ \ldots \circ \tau_1)(\sigma_{\text{init}})$ and because $\varepsilon \cdot h \in \sigma_{\text{init}}$, it follows that $(\tau_n \circ \ldots \circ \tau_1)(\varepsilon \cdot h) \in (\tau_n \circ \ldots \circ \tau_1)(\sigma_{\text{init}})$ and therefore $(\tau_n \circ \ldots \circ \tau_1)(\varepsilon \cdot h) \in \text{Repair}(\sigma')$.

Finally from lines 20 and 21 a new nexthop is added to the set of nexthops that contains the node u_1 where $u_1.\text{loc} = \text{tgt}(\text{edge})$ and $u_1.\text{ec} = \mathcal{H}_{j_k}$ and $u_1.\text{stack} = \sigma'$. Line 23 iterates over the nexthops and calls Dfs on line 28 with this new node.

To complete the proof, we put together the pieces to show that the 4 conditions above hold.

- line 28 calls $\text{Dfs}(\mathcal{N}, P, \_, u_1, i)$ for the u_1 described previously
- we know that $u_1.\text{loc} = \text{tgt}(\text{edge}) = v_1$
- we know that $\sigma_1 \in u_1.\text{stack}$ because $\sigma_1 \in \sigma'$ and $\sigma' = u_1.\text{stack}$
- we know that $\text{top}(\sigma_1) \in u_1.\text{ec}$ because $\text{top}(\sigma_1) \in \mathcal{H}_{j_k}$ and $u_1.\text{ec} = \mathcal{H}_{j_k}$

□

Corollary 5.1 (Property checking). *If i is the smallest step such that $\mathcal{N} \otimes \langle i, \ell \rangle$ then Algorithm 1 checks $P(u)$ for some DFS node u such that* $\text{top}([\![\mathcal{N}]\!]_i \ell) \in u$.

Proof. The proof follows directly from Theorem 5.1. At the $i-1$ step, we know that there must have been a call to $\text{Dfs}(\mathcal{N}, P, \_, u, i-1)$ for some u such that $\text{top}([\![\mathcal{N}]\!]_{i-1} \ell) \in u$ From the proof we can see that the algorithm will proceed to line 14, where it will check $P(u)$. □

Theorem 5.2 (Loop conditions). *Given a network $\mathcal{N}$ over $\mathcal{H}^*$, an input ℓ induces a loop if and only if there exists $i, k \in \mathbb{N}$, $\sigma, \sigma' \in \mathcal{H}^*$, and $h \in \mathcal{H}$ such that:*

$$\begin{array}{ll} (1) & \text{top}([\![\mathcal{N}]\!]_i \ell) = \langle u, \sigma \cdot h \rangle \\ (2) & \text{top}([\![\mathcal{N}]\!]_k \ell) = \langle u, \sigma \cdot \sigma' \cdot h \rangle \\ (3) & \forall j,\ i < j < k \Rightarrow \exists v, \sigma'',\ \text{top}([\![\mathcal{N}]\!]_j \ell) = \langle v, \sigma \cdot \sigma'' \rangle \end{array}$$

Proof. First, we require that no rule transformations τ ever both pop and push in the same transformation. For instance, the transformation $\tau_{\text{pop}} \circ \tau_{\text{push}}$ is disallowed, whereas $\tau_{\text{push}} \circ \tau_{\text{push}}$ is allowed. Note that this does not change the expressive power of KATRA since one can always separate such a transformation into multiple transformations across nodes to get the same effect.

Sufficient ($\Leftarrow$) Assume that the conditions (1), (2), and (3) above hold. We must prove that ℓ induces a loop. From (1) and (2), we know that there is a trace for $[\![N]\!]_k \ell$ to step k of the form:

$$\underbrace{\langle u_1, \sigma_1 \rangle}_{\text{step } 1} \to \underbrace{\langle u_2, \sigma_2 \rangle}_{\text{step } 2} \to \ldots \to \langle u_{i-1}, \sigma_{i-1} \rangle \to \underbrace{\langle u, \sigma \cdot h \rangle}_{\text{step } i} \to \underbrace{\langle u_{i+1}, \sigma_{i+1} \rangle \to \langle u_{i+2}, \sigma_{i+2} \rangle \to \ldots}_{\text{steps } i<j<k} \to \underbrace{\langle u, \sigma \cdot \sigma' \cdot h \rangle}_{\text{step } k}$$

We observe that from (1), (2), (3), the stack retains the prefix σ for all steps between i and k. From the assumption that transformations don't both push and pop the stack, and our model requirement that transformations can only match the top of the stack, this means that the forwarding for the stack at these steps does not depend on σ, and thus forall σ the subtrace starting at step i:

$$\underbrace{\langle u, \sigma \cdot h \rangle}_{\text{step } i} \to \underbrace{\langle u_{i+1}, \sigma_{i+1} \rangle \to \langle u_{i+2}, \sigma_{i+2} \rangle \to \ldots}_{\text{steps } i<j<k} \to \underbrace{\langle u, \sigma \cdot \sigma' \cdot h \rangle}_{\text{step } k}$$

would be the same for any such σ. For this reason, expanding out the trace from k steps to $2k-i$ steps, we observe the following continuation of the original trace:

$$\underbrace{\langle u, \sigma \cdot h \rangle}_{\text{step } i} \to \underbrace{\langle u_{i+1}, \sigma_{i+1} \rangle \to \langle u_{i+2}, \sigma_{i+2} \rangle \to \ldots}_{\text{steps } i<j<k} \to \underbrace{\langle u, \sigma \cdot \sigma' \cdot h \rangle}_{\text{step } k} \to \ldots \to \underbrace{\langle u, \sigma \cdot \sigma' \cdot \sigma'' \cdot h \rangle}_{\text{step } 2k-i}$$

In other words, because the forwarding between steps i and k did not depend on σ, it similarly will not depend on $(\sigma \cdot \sigma')$ for the same top of stack h between steps k and $k + (k-i) = 2k-i$ for the same loop interval. Moreover, we know that $\sigma'' = \sigma'$. This same reasoning applies inductively with the new prefix $(\sigma \cdot \sigma' \cdot \sigma')$. Thus we have an infinite loop.

Necessary ($\Rightarrow$) Let us assume there is an input ℓ that induces a loop in the network $\mathcal{N}$. We must prove that there exist $i, k \in \mathbb{N}$ and $\sigma, \sigma' \in \mathcal{H}^*$ and $h \in \mathcal{H}$ such that conditions (1), (2), and (3) hold. By way of contradiction, we assume ℓ induces a loop in $\mathcal{N}$ but that no such i, k, σ, σ', h exist to satisfy (1-3). Because the input ℓ induces a loop, we know that there is an infinite trace:

$$\langle u_1, \sigma_1 \rangle \to \underbrace{\langle u_2, \sigma_2 \rangle}_{t_1} \to \langle u_3, \sigma_3 \rangle \to \langle u_4, \sigma_4 \rangle \to \underbrace{\langle u_5, \sigma_5 \rangle}_{t_2} \to \langle u_6, \sigma_6 \rangle \to \ldots$$

Because the set of headers $\mathcal{H}$ comprising the hops in $\mathcal{H}^*$ is itself finite and because there is a permanent loop, there must be an infinite number of time steps $t_1, t_2, t_3, \ldots$ where the stack never goes below the size at time t_i in the future – i.e., $\forall j,\ j \geq t_i \Rightarrow |\sigma_{t_i}| \leq |\sigma_j|$. If there were no such infinite sequence, then there could not be a permanent loop since at some point t^*, the stack would have to continue to shrink forever ($\forall j_1,\ j_1 \geq t^* \Rightarrow \exists j_2,\ j_2 > j_1 \wedge |\sigma_{j_1}| < |\sigma_{j_2}|$) and would eventually become empty since stacks are finite. This would contradict the fact that there is a permanent loop since the packet would eventually be dropped when the stack becomes ε.

From the sequence of $t_1, t_2, t_3, \ldots$ and the finiteness of the topology, eventually there must eventually be a subset of t_i which we will call $t_{m_1}, t_{m_2}, t_{m_3} \ldots$ that repeat at the same node with the same top of stack:

$$\langle u_1, \sigma_1 \rangle \to \underbrace{\langle u_2, \sigma_2 \rangle}_{t_1} \to \langle u_3, \sigma_3 \rangle \to \langle u_4, \sigma_4 \rangle \to \underbrace{\langle u_5, \sigma_5 \rangle}_{t_2} \to \langle u_6, \sigma_6 \rangle \to \ldots \to \underbrace{\langle u_i, \sigma_i \rangle}_{t_{m_1}} \to \ldots \to \underbrace{\langle u_k, \sigma_k \rangle}_{t_{m_2}} \to \ldots$$

where $u_{t_{m_1}} = u_{t_{m_2}}$, and $\text{top}(\sigma_{t_{m_1}}) = \text{top}(\sigma_{t_{m_2}})$ and so on for all t_{m_i}. Because we know that at time t_{m_1} the stack $\sigma_{t_{m_1}}$ never again goes below this size, if $\sigma_{t_{m_1}} = \sigma \cdot h$, then every stack in the trace from this time on must start with σ. The earliest two times t_{m_1} and t_{m_2} capture exactly i, k in the theorem, and $\sigma_{t_{m_1}}$ captures $\sigma \cdot h$ (condition 1). The trace retains the prefix σ after time t_{m_1} (conditions 2, 3). And the nodes and top of stacks are the same at each time t_{m_i} being h (condition 2). □