KATch: A Fast Symbolic Verifier for NetKAT*

MARK MOELLER†, Cornell University, USA
JULES JACOBS†, Cornell University, USA
OLIVIER SAVARY BELANGER, Galois, USA
DAVID DARAIS, Galois, USA
COLE SCHLESINGER, Galois, USA
STEFFEN SMOLKA, Google, USA
NATE FOSTER, Cornell University, USA
ALEXANDRA SILVA, Cornell University, USA

We develop new data structures and algorithms for checking verification queries in NetKAT, a domain-specific language for specifying the behavior of network data planes. Our results extend the techniques obtained in prior work on symbolic automata and provide a framework for building efficient and scalable verification tools. We present KATch, an implementation of these ideas in Scala, featuring an extended set of NetKAT operators that are useful for expressing network-wide specifications, and a verification engine that constructs a bisimulation or generates a counter-example showing that none exists. We evaluate the performance of our implementation on real-world and synthetic benchmarks, verifying properties such as reachability and slice isolation, typically returning a result in well under a second, which is orders of magnitude faster than previous approaches. Our advancements underscore NetKAT's potential as a practical, declarative language for network specification and verification.

CCS Concepts: • **Theory of computation → Formal languages and automata theory**.

Additional Key Words and Phrases: NetKAT, network verification, program equivalence, symbolic automata.

ACM Reference Format:
Mark Moeller, Jules Jacobs, Olivier Savary Belanger, David Darais, Cole Schlesinger, Steffen Smolka, Nate Foster, and Alexandra Silva. 2024. KATch: A Fast Symbolic Verifier for NetKAT. *Proc. ACM Program. Lang.* 8, PLDI, Article 224 (June 2024), 24 pages. https://doi.org/10.1145/3656454

1 INTRODUCTION

In the automata-theoretic approach to verification, programs and specifications are each encoded as automata, and verification tasks are reduced to standard questions in formal language theory—e.g.,

*This research was developed with funding from the Defense Advanced Research Projects Agency (DARPA) and the Office of Naval Research (ONR). The views, opinions, and/or findings contained in this material are those of the authors and should not be interpreted as representing the official views or policies of the Department of the Navy, DARPA, or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation hereon. DISTRIBUTION A. Approved for public release; distribution unlimited.
†Equal contribution.

Authors' Contact Information: Mark Moeller, Cornell University, Ithaca, NY, USA, mdm367@cornell.edu; Jules Jacobs, Cornell University, Ithaca, NY, USA, jj758@cornell.edu; Olivier Savary Belanger, Galois, Portland, OR, USA, olivier@galois.com; David Darais, Galois, Portland, OR, USA, darais@galois.com; Cole Schlesinger, Galois, Portland, OR, USA, coles@galois.com; Steffen Smolka, Google, Mountain View, CA, USA, smolkaj@google.com; Nate Foster, Cornell University, Ithaca, NY, USA, jnfoster@cs.cornell.edu; Alexandra Silva, Cornell University, Ithaca, NY, USA, alexandra.silva@cornell.edu.

This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2024 Copyright held by the owner/author(s).
ACM 2475-1421/2024/6-ART224
https://doi.org/10.1145/3656454

membership, emptiness, containment, etc. [44]. The approach became popular in the mid-1980s, driven by use of temporal logics and model checking in hardware verification, and it remains a powerful tool today. In particular, automata provide natural models of phenomena like transitive closure, which arises often in programs but is impossible to express in pure first-order logic.

NetKAT, a domain-specific language for specifying and verifying the behavior of network data planes, is a modern success story for the automata-theoretic approach. NetKAT programs denote sets of traces (or "histories") where a trace is the list of packets "seen" at each hop in the network. The NetKAT language specifies these traces using a regular expression-like syntax, as follows:

$$p, q ::= \bot \mid \top \mid f = v \mid f \neq v \mid f \leftarrow v \mid \mathrm{dup} \mid p + q \mid p \cdot q \mid p^\star$$

Unlike ordinary regular expressions, which are stateless, NetKAT programs manipulate state in packets. Accordingly, NetKAT's atoms are not letters, but *actions* that either drop or forward the current packet ($\bot$ or $\top$), modify a header field ($f \leftarrow v$), test a header field against a value ($f = v$, $f \neq v$), or append the current packet to the trace (dup). The regular expression operations copy and forward a packet to two different programs ($p + q$), sequence two programs ($p \cdot q$), or loop a program ($p^\star \equiv \top + p + p \cdot p + p \cdot p \cdot p + \cdots$).

A NetKAT program thus gives a declarative specification of the network's global behavior in terms of sets of traces. Specifically, we can model the location of a packet in the network using a special header field (usually named sw for "switch"), that we can update ($\mathrm{sw} \leftarrow \ell$) to logically move the packet from its current location to a new location ℓ. Given a declarative description of the intended behavior, the NetKAT compiler generates a set of local forwarding tables for individual switches that together realize the global behavior [20, 37].

Moreover, NetKAT not only plays a role as an implementation language, but also as a language for expressing verification queries, analogous to verification tools powered by SMT solvers [3, 31, 43]. In particular, as NetKAT includes a union operator ("+"), containment can be reduced to program equivalence—i.e., $p \sqsubseteq q$ if and only if $p + q \equiv q$. Hence, unlike other contemporary tools, which rely on bespoke encodings and algorithms for checking network properties like reachability, slice isolation, etc. [19, 26, 27, 46], NetKAT allows a wide range of practical verification questions to be answered using the same foundational mechanism, namely program equivalence. For example,

- *Network Reachability:* Are there any packets that can go from location 121 to 543 in the network specified by the NetKAT program net? Formally: $(\mathrm{sw} \leftarrow 121) \cdot \mathrm{net}^\star \cdot (\mathrm{sw} = 543) \stackrel{?}{\equiv} \bot$
- *Slice Isolation:* Are slices net_1 and net_2 logically disjoint, even though their implementation on shared infrastructure uses the same devices? Formally: $\mathrm{net}_1^\star + \mathrm{net}_2^\star \stackrel{?}{\equiv} (\mathrm{net}_1 + \mathrm{net}_2)^\star$

NetKAT's dup primitive is a key construct that enables reducing a wide range of network-specific properties to program equivalence. Recall that dup appends the headers of the current packet to the trace. Hence, to verify properties involving the network's internal behavior, we add intermediate packets to the trace using dup, making them relevant for program equivalence. Conversely, if we only care about the input-output behavior of the entire network, we can omit dup from the specification, so intermediate packets are not considered by the check for program equivalence.

The story so far is appealing, but there is a major fly in the ointment. To decide program equivalence, the automata-theoretic approach relies on the translation from NetKAT programs to automata. But standard NetKAT automata have an enormous space of potential transitions—i.e., the "alphabet" has a "character" for every possible packet. So using textbook algorithms for building automata and checking equivalence would clearly be impractical. In fact, NetKAT equivalence is PSPACE-complete [1]. Instead, what's needed are symbolic techniques for encoding the state space and transition structure of NetKAT automata that avoid combinatorial blowup in the common case.

Prior work on symbolic automata provides a potential solution to the problems that arise when working with large (or even infinite) alphabets. The idea is to describe the transitions in the automata using logical formulas [16, 17] or Binary Decision Diagrams (BDDs) [36] rather than concrete characters. Algorithms such as membership, containment, and equivalence can then be reformulated to work with these symbolic representations. Symbolic automata have been successfully applied to practical problems such as building input sanitizers for Unicode, which has tens of thousands of characters [24]. However, NetKAT's richer semantics precludes the direct adoption of standard notions of symbolic automata. In particular, NetKAT's transitions describe not only predicates but also transformations on the current packet. As Pous writes, it "seems feasible" to generalize his work to NetKAT, but "not straightforward" [36].

In this paper, we close this gap and develop symbolic techniques for checking program equivalence for NetKAT. In doing so, we address three key challenges:

Challenge 1: Expressive but Compact Symbolic Representations. The state space and transitions of NetKAT automata are very large due to the way the "alphabet" is built from the space of all possible packets. Orthogonally, these characters also encode packet transformations, as reflected in NetKAT's packet-processing semantics. It is crucial that the symbolic representations for NetKAT programs be compact and admit efficient equivalence checks.

Challenge 2: Extended Logical Operators. In the tradition of regular expressions, NetKAT only includes sequential composition ($e_1 \cdot e_2$) and union operators ($e_1 + e_2$)—the latter computes the union of the sets of traces described by e_1 and e_2. However, when verifying network-wide properties it is often useful to have operators for intersection ($e_1 \cap e_2$), difference ($e_1 \setminus e_2$), and symmetric difference ($e_1 \oplus e_2$). Having native support for these operators at the level of syntax and in equivalence checking algorithms allows the reduction of all verification queries to an emptiness check—e.g., $A \equiv B$ reduces to $A \oplus B \equiv \bot$, and $A \sqsubseteq B$ reduces to $A \setminus B \equiv \bot$.

Challenge 3: Support for Counter-Examples. When an equivalence check fails, it can be hard to understand the cause of the failure, and which changes might be needed to resolve the problem. It is therefore important to be able to construct *symbolic counter-examples* that precisely capture the input packets and traces that cause equivalence to fail.

In addressing the above challenges, we make the following technical contributions:

Contribution 1: Efficient symbolic representations *(Section 3).* We design a symbolic data structure called Symbolic Packet Programs (SPPs) for representing both sets of packets and transformations on packets in a symbolic manner. SPPs generalize classic BDDs and are asymptotically more efficient than the representations used in prior work on NetKAT. In particular, SPPs support efficient sequential composition ($e_1 \cdot e_2$) and can be made *canonical* which, with hash consing, allows us to check equivalence of SPPs in constant time.

Contribution 2: Symbolic Brzozowski derivatives *(Section 4).* We introduce a new form of symbolic Brzozowski derivative to produce automata for NetKAT programs. Our approach naturally and efficiently supports the extended "negative" logical operators mentioned above, in contrast to prior approaches that only support "positive" operators like union.

Contribution 3: Symbolic bisimilarity checking *(Section 5).* Building on the foundation provided by SPPs and deterministic NetKAT automata, we develop symbolic algorithms for checking bisimilarity. We present a symbolic version of the standard algorithm that searches in the forward direction through the state space of the automata under consideration. We also present a novel

Syntax		Description	Semantics
$p, q ::=$			$[\![p]\!](\alpha : \mathsf{Pk}) : \mathcal{P}(\mathsf{Pk}^+) =$
|	$\bot$	*False*	$\emptyset$
|	$\top$	*True*	$\{\alpha\}$
|	$f = v$	*Test equals*	if $\alpha_f = v$ then $\{\alpha\}$ else $\emptyset$
|	$f \neq v$	*Test not equals*	if $\alpha_f \neq v$ then $\{\alpha\}$ else $\emptyset$
|	$f \leftarrow v$	*Modification*	$\{\alpha[f \leftarrow v]\}$
|	dup	*Duplication*	$\{\alpha\alpha\}$
|	$p + q$	*Union*	$[\![p]\!](\alpha) \cup [\![q]\!](\alpha)$
|	$p \cdot q$	*Sequencing*	$\{\mathbf{ab} \mid \mathbf{a}\beta \in [\![p]\!](\alpha),\ \mathbf{b} \in [\![q]\!](\beta)\}$
|	$p^\star$	*Iteration*	$\bigcup_{n \geq 0} [\![p^n]\!]$

Values $v ::= 0 \mid 1 \mid \ldots \mid n$ Fields $f ::= f_1 \mid \ldots \mid f_k$ Packets $\alpha ::= \{f_1 = v_1, \ldots, f_k = v_k\}$

Fig. 1. NetKAT syntax and semantics.

backward algorithm that computes symbolic counter-examples to equivalence—i.e., the precise set of input packets for which equivalence fails.

***Contribution 4: KATch implementation** (Section 6) **and evaluation** (Section 7).* Finally, we present a new verification tool, called KATch, implemented in Scala. KATch implements symbolic bisimilarity checking, including an extended set of extended logical operators, as well as symbolic counter-examples. We evaluate the performance of KATch on a variety of real-world topologies, and show that it efficiently answers a variety of verification queries and scales to much larger networks than prior work. Due to its use of our efficient data structures, KATch is several orders of magnitude faster than prior implementations of NetKAT on these realistic examples. Moreover, KATch is faster than prior work on synthetic combinatorial benchmarks by arbitrary large factors.

2 NETKAT EXPRESSIONS AND AUTOMATA

This section reviews basic definitions for NetKAT to set the stage for the new contributions presented in the subsequent sections. NetKAT is a language for specifying the packet-forwarding behavior of network data planes [1, 22, 37].[1] The syntax and semantics of NetKAT is presented in Figure 1, and is based on Kozen's Kleene Algebra with Tests (KAT) [29].

To a first approximation, NetKAT can be thought of as a simple, imperative language that operates over packets, where a *packet* is a finite record assigning values to fields. The basic primitives in NetKAT are packet tests ($f = v$, $f \neq v$) and packet modifications ($f \leftarrow v$). Program expressions are then compositionally built from tests and packet modifications, using union (+), sequencing ($\cdot$), and iteration ($\star$). Conditionals and loops can be encoded in the standard way:

$$\textbf{if } b \textbf{ then } p \textbf{ else } q \triangleq b \cdot p + \neg b \cdot q \qquad \textbf{while } b \textbf{ do } p \triangleq (b \cdot p)^\star \cdot \neg b.$$

In a network, conditionals can be used to model the behavior of the forwarding tables on individual switches while iteration can be used to model the iterated processing performed by the network as a whole; the original paper on NetKAT provides further details [1]. Note that assignments and tests

[1]Networks have a control plane, which computes paths through the topology using distributed routing protocols (or a software-defined networking controller), and a data plane, which implements paths using high-speed hardware and software pipelines. NetKAT models the behavior of the latter.

in NetKAT are always against constant values. This is a key restriction that makes equivalence decidable and aligns with the capabilities of data plane hardware. The dup primitive makes a copy of the current packet and appends it to the trace, which only ever grows as the packet goes through the network. This primitive is crucial for expressing network-wide properties, as it allows the semantics to "see" the intermediate packets processed on internal switches.

NetKAT's formal semantics, given in Figure 1, defines the action of a program on an input packet α, producing a set of output traces. The semantics for $\top$ gives a single trace containing the single input packet α, whereas $\bot$ produces no traces. Tests ($f = v$ and $f \neq v$) produce a singleton trace or no traces, depending on whether the test succeeds or fails. Modifications ($f \leftarrow v$) produce a singleton trace with the modified packet. Duplication (dup) produces a single trace with two copies of the input packet. Union ($p + q$) produces the union of the traces produced by p and q. Sequential composition ($p \cdot q$) produces the concatenation of the traces produced by p and q, where the last packet in output traces of p is used as the input to q. Finally, iteration ($p^\star$) produces the union of the traces produced by p iterated zero or more times. Consider the following example:

$$(x=0 \cdot x \leftarrow 1 \cdot \text{dup} \;+\; x=1 \cdot x \leftarrow 0 \cdot \text{dup})^\star$$

This program repeatedly flips the value of x between 0 and 1, and traces the packet at each step. On input packet $x=0$, it produces the following traces:

$$\{[x=0], [x=0, x=1], [x=0, x=1, x=0], [x=0, x=1, x=0, x=1], \ldots\}$$

Consider what happens if we change the program to $(x=0 \cdot x \leftarrow 1 \cdot \text{dup} \;+\; x \leftarrow 0 \cdot \text{dup})^\star$. Unlike the previous example, where the tests were disjoint, this program can generate multiple outputs for a given input packet, as the right branch of the union can always be taken, leading to sequences $x=0, x=0, \ldots$. On the other hand, if the $x \leftarrow 1$ assignment is performed, then the left branch cannot be taken the next time, because the test $x=0$ will fail. Hence, the program produces traces with sequences of $x=0$ packets, with singular $x=1$ packets between them.

As these simple examples show, despite the fact that assignments and tests in NetKAT programs are always against constant values, the behavior of NetKAT programs can be complicated, particularly when conditionals, iteration, and multiple packet fields are involved.

Traditional presentations of NetKAT distinguish a syntactic test fragment that includes negation:

$$t, r ::= \bot \mid \top \mid f=v \mid t \vee r \mid t \wedge r \mid \neg t \qquad \text{(Test fragment)}$$

$$p, q ::= t \mid f \leftarrow v \mid p + q \mid p \cdot q \mid \text{dup} \mid p^\star \qquad \text{(General expressions)}$$

This distinction is necessary because negation $\neg p$ only makes sense on the test fragment. We replace general negation $\neg t$ in our presentation with only negated field tests $f \neq v$. There is no loss of generality—we translate a test fragment expression t to $[t]_0$ using the DeMorgan rules:

$$[\bot]_0 = \bot \qquad [t \vee r]_0 = [t]_0 + [r]_0 \qquad [\bot]_1 = \top \qquad [t \vee r]_1 = [t]_1 \cdot [r]_1$$

$$[\top]_0 = \top \qquad [t \wedge r]_0 = [t]_0 \cdot [r]_0 \qquad [\top]_1 = \bot \qquad [t \wedge r]_1 = [t]_1 + [r]_1$$

$$[f=v]_0 = (f=v) \qquad [\neg t]_0 = [t]_1 \qquad [f=v]_1 = (f \neq v) \qquad [\neg t]_1 = [t]_0$$

Conditional dup, equivalence, and subsumption. An interesting feature of NetKAT is that dup need not appear the same number of times on all paths through the expression. Consider the program $(f \leftarrow v + f \leftarrow w + \text{dup})^\star$, which either sets field f to v or to w, or logs the packet to the trace, and then repeats. Any number of writes can happen between two dups, and only the effect of the last write before the dup gets logged to the trace. This is allowed and fully supported by KATch.

The use of dup controls the type of equivalence that is checked. If we insert a dup after every packet field write, then we obtain trace equivalence. If we insert no dups whatsoever, we obtain

Input: A pair of NetKAT automata $(S, s_0, \delta, \epsilon)$, $(S', s_0', \delta', \epsilon')$.
Returns: A Boolean indicating whether the automata are equivalent.
$W \leftarrow \{(s_0, s_0', \alpha) \mid \alpha \in \mathsf{Pk}\}$;
while W changes **do**
  for $(s, s', \alpha) \in W$ **do**
    if $\epsilon(s, \alpha) \neq \epsilon'(s', \alpha)$ **then return false**;
    $W \leftarrow W \cup \{(\delta(s, \alpha)(\alpha'), \delta'(s', \alpha)(\alpha'), \alpha') \mid \alpha' \in \mathsf{Pk}\}$
return true;

Fig. 2. Bisimulation algorithm à la Hopcroft & Karp [25]

input-output equivalence. By inserting dups in some places but not others, we can control the equivalence that is checked in a fine-grained manner.

In addition to $p \equiv q$, we may want to check inclusion $p \sqsubseteq q$. This can be expressed as $p + q \equiv q$ or equivalently, in KATch, as $p \setminus q \equiv \bot$. As before, the number of dups controls the type of inclusion that is checked: from trace inclusion to input-output inclusion, or something in between.

2.1 NetKAT Automata

NetKAT's semantics induce an equivalence $(p \equiv q) \triangleq (\llbracket p \rrbracket = \llbracket q \rrbracket)$ on syntactic programs. This equivalence is decidable, and can be computed by converting programs to automata and checking for automata equivalence. To convert NetKAT programs to automata, the standard approach is to use Antimirov derivatives. We do not give the details here, but the interested reader can find them in [22], and for our symbolic automata, in Section 4. We continue with a brief overview of NetKAT automata and how to check their equivalence. A NetKAT automaton $(S, s_0, \langle \epsilon, \delta \rangle)$ consists of a set of states S, a start state s_0, and a pair of functions that depend on an input packet:

$$\epsilon : S \times \mathsf{Pk} \to 2^{\mathsf{Pk}} \qquad \delta : S \times \mathsf{Pk} \to S^{\mathsf{Pk}}$$

Intuitively, the observation function ϵ is analogous to the notion of a final state, and models the output packets produced from an input packet at a given state. The transition function δ models the state transitions that can occur when processing an input packet at a given state. Because transitions can modify the input packet, the transition structure $\delta(s, \alpha)$ is itself a function of the modified packet, and tells us to transition to state $\delta(s, \alpha)(\alpha')$ when input packet α is modified to α'.

In terms of the semantics, a transition in the automaton corresponds to executing to the next dup in a NetKAT program, and appends the current packet to the trace. It should also be noted that states in the automaton do not necessarily correspond to network devices; because the location of the packet is modeled as just a field in the packet, there can be states in the automaton that handle packets from multiple devices, and the packets of a device can be handled by multiple states.

Carry-on packets. Unlike traditional regular expressions and automata, NetKAT is stateful, and the output packet of a transition is carried on to the next state. This makes NetKAT fundamentally different from traditional regular expressions, resulting in challenges in the semantics (which is not a straightforward trace or language semantics) and in designing equivalence procedures, as they have to account for the carry-on packet. Technically it would be possible to fold the packet into the state of the automaton and approach the semantics and the problem of checking equivalence using classical automata. However, because the number of possible packets is exponential in the number of fields and values, such conversion results in enormous state blowup and is therefore impractical.

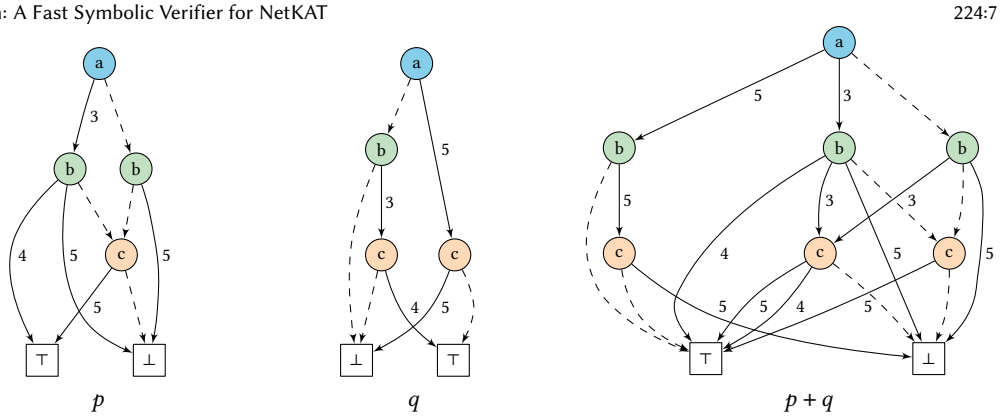

Fig. 3. Examples of SPs, where $p \triangleq (a{=}3 \cdot b{=}4 + b{\neq}5 \cdot c{=}5)$ and $q \triangleq (b{=}3 \cdot c{=}4 + a{=}5 \cdot c{\neq}5)$.

2.2 Bisimulation of NetKAT Automata

NetKAT automata can be used to decide $p \equiv q$. Intuitively, while traces may be infinite, transitions only depend on the current packet, which has a finite number of distinct possible values.

Given two automata $(S, s_0, \langle \epsilon, \delta \rangle)$ and $(S', s_0', \langle \epsilon', \delta' \rangle)$, we check their equivalence by considering all possible input packets separately. We run the two automata in parallel, starting from an input packet α at their respective start states. We first check that the immediate outputs $\epsilon(s_0, \alpha)$ and $\epsilon'(s_0', \alpha)$ are the same. If so, we check that the transitions $\delta(s_0, \alpha)$ and $\delta'(s_0', \alpha)$ are equivalent, by recursively checking the equivalence of the states $\delta(s_0, \alpha)(\alpha')$ and $\delta'(s_0', \alpha)(\alpha')$ for all α'.

We can implement this strategy using a work list algorithm. The work list contains triples (s, s', α), where s and s' are states in the two automata, and α is an input packet. Initially, the work list contains (s_0, s_0', α) for all packets α. We then repeatedly take triples (s, s', α) from the work list, and check that $\epsilon(s, \alpha) = \epsilon'(s', \alpha)$ and add $(\delta(s, \alpha)(\alpha'), \delta'(s', \alpha)(\alpha'), \alpha')$ to the work list for all α'. If at any point we find that $\epsilon(s, \alpha) \neq \epsilon'(t, \alpha)$, then the automata are not equivalent. If the work list stabilizes, the automata are equivalent. This algorithm is shown in Figure 2.

Of course, the issue with this algorithm is that the space of packets is huge. The rest of this paper develops techniques for working with symbolic representations of packets and automata, allowing us to represent and manipulate large sets of packets and to efficiently check equivalence of automata without explicitly enumerating the space of packets.

3 SYMBOLIC NETKAT REPRESENTATIONS

In this section, we develop symbolic techniques to trace an entire set of packets at once, vastly reducing the number of iterations required to compute a bisimulation in practice. These techniques will allow us to revise the approach of Figure 2 with a much more efficient version in Section 5.

First, we introduce a representation for symbolic packets, which represent sets of packets, or equivalently, the fragment of NetKAT where all atoms are tests. This representation is essentially a natural n-ary variant of binary decision diagrams (BDDs) [12]. Second, we introduce Symbolic Packet Programs (SPPs), a new representation for symbolic transitions in NetKAT automata, representing the dup-free fragment of NetKAT—i.e., the fragment in which all atoms are tests or assignments. The primary challenge is to design this representation so that it is efficiently closed under the NetKAT operations. Furthermore, we need to be able to efficiently compute the transition of a symbolic packet over a SPP, both forward and backward.

Primitive tests:

$(f{=}v) \triangleq \mathsf{SP}(f, \{v \mapsto \top\}, \bot) \qquad (f{\neq}v) \triangleq \mathsf{SP}(f, \{v \mapsto \bot\}, \top)$

Operations, base cases:

$$\begin{array}{llll} \top \mathbin{\hat{+}} \top \triangleq \top & \top \mathbin{\hat{:}} \top \triangleq \top & \top \mathbin{\hat{\cap}} \top \triangleq \top & \top \mathbin{\hat{\oplus}} \top \triangleq \bot \quad \top \mathbin{\hat{-}} \top \triangleq \bot \\ \top \mathbin{\hat{+}} \bot \triangleq \top & \top \mathbin{\hat{:}} \bot \triangleq \bot & \top \mathbin{\hat{\cap}} \bot \triangleq \bot & \top \mathbin{\hat{\oplus}} \bot \triangleq \top \quad \top \mathbin{\hat{-}} \bot \triangleq \top \\ \bot \mathbin{\hat{+}} \top \triangleq \top & \bot \mathbin{\hat{:}} \top \triangleq \bot & \bot \mathbin{\hat{\cap}} \top \triangleq \bot & \bot \mathbin{\hat{\oplus}} \top \triangleq \top \quad \bot \mathbin{\hat{-}} \top \triangleq \bot \\ \bot \mathbin{\hat{+}} \bot \triangleq \bot & \bot \mathbin{\hat{:}} \bot \triangleq \bot & \bot \mathbin{\hat{\cap}} \bot \triangleq \bot & \bot \mathbin{\hat{\oplus}} \bot \triangleq \bot \quad \bot \mathbin{\hat{-}} \bot \triangleq \bot \end{array}$$

Operations, inductive case:

$$\mathsf{SP}(f, b_p, d_p) \mathbin{\hat{\pm}} \mathsf{SP}(f, b_q, d_q) \triangleq \mathsf{sp}(f, b', d_p \mathbin{\hat{\pm}} d_q)$$
$$\text{where } b' = \{v \mapsto b_p(v; d_p) \mathbin{\hat{\pm}} b_q(v; d_q) \mid v \in \mathsf{dom}(b_p \cup b_q)\}$$

Expansion:

$$p \equiv \mathsf{SP}(f, \emptyset, p) \qquad \text{if} \qquad p \in \{\top, \bot, \mathsf{SP}(f', b, d)\} \text{ where } f \sqsubset f'$$

Smart constructor:

$$\mathsf{sp}(f, b, d) \triangleq \text{ if } b' = \emptyset \text{ then } d \text{ else } \mathsf{SP}(f, b', d)$$
$$\text{where } b' \triangleq \{v \mapsto p \mid v \mapsto p \in b, p \neq d\}$$

Fig. 4. Definition of the SP operations. The inductive case is identical for all operations (indicated by $\hat{\pm}$), and applies when both SPs test the same field. Expansion inserts a trivial SP node to reduce the remaining cases to the inductive case. The notation $b(v; d)$ means the child $b(v)$ if $v \in \mathsf{dom}(b)$, or the default case d otherwise.

3.1 Symbolic Packets

We begin by choosing a representation for symbolic packets. A symbolic packet $p \subseteq \mathsf{Pk}$ is a set of concrete packets, represented compactly as a decision diagram. Syntactically, symbolic packets are NetKAT expressions with atoms restricted to tests, represented in the following canonical form:

$$p \in \mathsf{SP} ::= \bot \mid \top \mid \underbrace{\mathsf{SP}(f,\ \{\ldots, v_i \mapsto q_i, \ldots\},\ q)}_{\equiv\ \sum_i f{=}v_i \cdot q_i\ +\ (\prod_i f{\neq}v_i) \cdot q}$$

That is, symbolic packets form an n-ary tree, where each child q_i is labeled with a test of the current field f, and the default case q is labeled with the negation of the other tests. Hence, a given concrete packet has a unique path through the tree.

The following conditions need to be satisfied for a symbolic packet to be in canonical form, inspired by the analogous properties of BDDs:

Reduced If a child q_i is equal to the default case q, it is removed. If only the default case remains, the symbolic packet is reduced to the default case itself.

Ordered A path down the tree always follows the same order of fields, and the children q_i are ordered by the value v_i.

These conditions ensure the representation of a symbolic packet is unique, in the sense that two symbolic packets are semantically equal if and only if they are syntactically equal.

Representation. As with BDDs, we share nodes in the tree, making the representation of a symbolic packet a directed acyclic graph, as shown in Figure 3. Vertices are labeled with the packet field they test. Solid arrows labeled with a number encode the test value, and dashed arrows represent a default case. The sinks of the graph are labeled with $\top$ or $\bot$, indicating membership in the set. On the left, we have a symbolic packet representing all packets where $a = 3$ and $b = 4$, or $b \neq 5$ and $c = 5$. In the middle, we have a symbolic packet representing all packets where $a = 3$ and $c = 4$, or $a = 5$ and $c \neq 5$. On the right, we have the union of these symbolic packets.

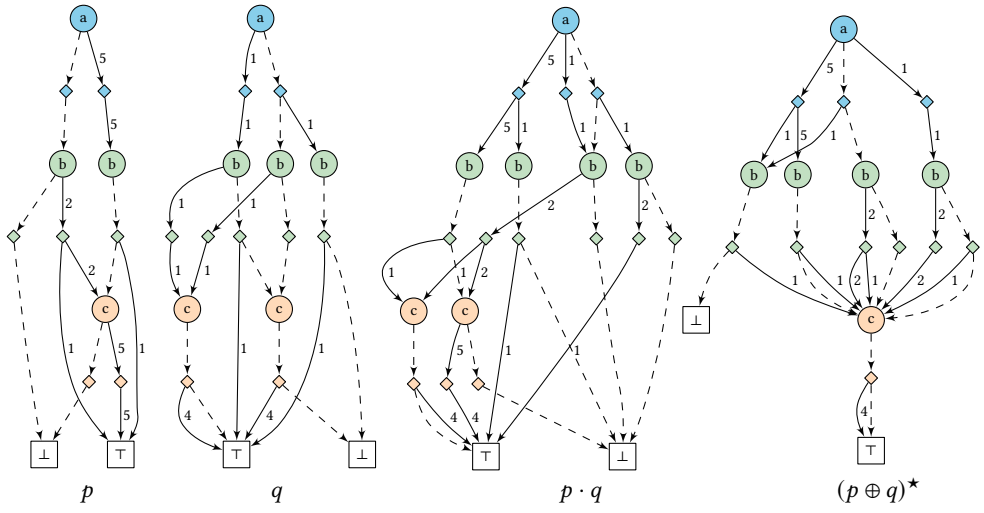

Fig. 5. Examples of SPPs, where $p \triangleq (a{=}5 + b{=}2) \cdot (b{\leftarrow}1 + c{=}5)$, and $q \triangleq (b{=}1 + c{\leftarrow}4 + a{\leftarrow}1 \cdot b{\leftarrow}1)$.

Operations. Symbolic packets are closed under the NetKAT operations:

$$\hat{+}, \hat{;}, \hat{\cap}, \hat{\oplus}, \hat{\neg} \ : \ \mathsf{SP} \times \mathsf{SP} \to \mathsf{SP} \qquad \hat{\star} \ : \ \mathsf{SP} \to \mathsf{SP} \qquad f{=}v, f \neq v \ : \ \mathsf{SP}$$

As defined in Figure 4, operations on symbolic packets are computed by traversing the two trees in parallel, recursively taking the operation of the children, making sure to maintain canonical form using the smart constructor. Repetition $p^\star \triangleq \top$ is trivial for symbolic packets. All other operations are defined in the extended version of this paper [33].

Specifications. Each of these operations is uniquely specified by two correctness conditions:

(1) They semantically match their counterpart: $[\![p \mathbin{\hat{+}} q]\!] = [\![p + q]\!]$ for all $p, q \in \mathsf{SP}$.
(2) They maintain canonical form: $p \mathbin{\hat{+}} q$ is reduced and ordered if p, q are.

For further formal treatment of SPs, see the extended version of this paper [33].

3.2 Symbolic Transitions

We now introduce a representation for symbolic transitions in NetKAT automata. Whereas symbolic packets represent the fragment of NetKAT where all atoms are tests, symbolic transitions correspond to the larger dup-free fragment, where all atoms are tests or assignments. This introduces additional challenges for a canonical representation, as well as for the operations. For instance, sequential composition is no longer equivalent to intersection, and the star operator is no longer trivial.

$$p \in \mathsf{SPP} ::= \bot \mid \top \mid \underbrace{\mathsf{SPP}(f, \{\ldots, v_i \mapsto \{\ldots, w_{ij} \mapsto q_{ij}, \ldots\}, \ldots\}, \{\ldots, w_i \mapsto q_i, \ldots\}, q)}_{\equiv \ \sum_i f{=}v_i \cdot \sum_j f{\leftarrow}w_{ij} \cdot q_{ij} \ + \ (\prod_i f{\neq}v_i) \cdot (\sum_i f{\leftarrow}w_i \ + \ (\prod_i f{\neq}w_i) \cdot q)}$$

Like SPs, SPPs have two base cases, $\top$ and $\bot$. Also like SPs, SPPs test a field f of the input packet against a series of values $v_0, \ldots, v_n$, with a default case $f \neq v_0 \cdots f \neq v_n$. However, instead of continuing recursively after the test, SPPs non-deterministically assign a value w_{ij} to the field f of the output packet, and continue recursively with the corresponding child q_{ij}. This way, SPPs can output more than one packet for a given input packet, and can also output packets with different

values for the same field. The default case is further split into two cases, one where the field f is non-deterministically assigned a new value (like in the other cases), and an identity case where the field f keeps the same value as the input packet. However, the packet for the latter case is not produced when the input packet's field f had a value that could also be produced by the explicit assignments in the default case. This ensures that for a given input packet, a given output packet is produced by a unique path through the SPP.

The following conditions need to be satisfied for a SPP to be in canonical form:

Reduced If a child q_{ij} is equal to $\bot$, it is removed (with the exception of the default-identity case, which is always kept). If one of the default-assignment cases for a value w is $\bot$, it is removed, but to keep the behavior equivalent, an additional test for the same value w is added, with an empty sequence of assignments (if a test for the value w was already present, nothing is added). Further, each of the non-default cases is analyzed, and if we determine that it behaves semantically like the default case for that input value, then it is removed. If only the default case remains, the SPP is reduced to the default case itself.

Ordered A path down the tree always follows the same order of fields, and both the tests and the assignments are ordered by the value v_i or w_{ij}.

These conditions ensure the representation of a SPP is unique, in the sense that two SPPs are semantically equal if and only if they are syntactically equal.

Representation. Like symbolic packets, we represent SPPs as a directed acyclic graph, with duplicate nodes shared. Examples of SPPs are shown in Figure 5. Vertices are labeled with the packet field that they test. Solid arrows labeled with a number represent the tests, and dashed arrows represent the default case. Each test is followed by a non-deterministic assignment of a new value to the field, indicated by the small diamonds. In the default case, the non-deterministic assignment also has an identity case (keeping the value of the field unchanged), indicated by a dashed arrow emanating from the diamond.

Consider the action of the first SPP in Figure 5 on the concrete packet $a=5 \cdot b=3 \cdot c=5$. The first test $a=5$ succeeds, and the field a is assigned the value 5. The b field is then tested, but the b node only has a default case. The default case does have a non-deterministic assignment, which can set the b field to the new value 1, or it can keep the old value 3. In case the new value of b is 1, the packet is immediately accepted by the $\top$ node, so the packet $a=5 \cdot b=1 \cdot c=5$ is produced. In case the old value of b is kept, the c field is tested, and in our case the value of the c field is 5. In this case, the value of the field is unchanged, and the packet $a=5 \cdot b=3 \cdot c=5$ is produced. In summary, for input packet $a=5 \cdot b=3 \cdot c=5$, the SPP produces packets $a=5 \cdot b=1 \cdot c=5$ and $a=5 \cdot b=3 \cdot c=5$.

When we sequentially compose the two SPPs on the left, we get the third SPP shown. In other words, if we take a concrete packet, and first apply the first SPP, and then apply the second SPP to all of the resulting packets, then we get the same result as if we apply the SPP on the right directly to the concrete packet. Sequential composition is a relatively complex operation, but algorithmically it is a key strength of our representation and reduced/ordered invariant. To understand why, consider taking a concrete packet, and applying the first SPP to it. Once we have applied the a-layer of the SPP to the packet, we already know what the value of the a field in the output packet will be. We can therefore immediately continue with the a layer of the second SPP, without having to consider the other layers of the first SPP. This is in contrast to a naive algorithm, which would have to consider the entire first SPP before being able to apply the second SPP. This property makes it possible to compute the sequential composition of two SPPs efficiently in practice, and is a key contributor to the performance and scalability of our system.

Primitive tests and mutation:

$(f=v) \triangleq \mathsf{SPP}(f, \{v \mapsto \{v \mapsto \top\}\}, \emptyset, \bot)$ $\quad$ $(f \leftarrow v) \triangleq \mathsf{SPP}(f, \emptyset, \{v \mapsto \top\}, \bot)$

$(f \neq v) \triangleq \mathsf{SPP}(f, \{v \mapsto \{v \mapsto \bot\}\}, \emptyset, \top)$

Operations, base cases:

Identical to those in Figure 4.

Operations, inductive cases:

$$\mathsf{SPP}(f, b_p, m_p, d_p) \mathbin{\hat{+}} \mathsf{SPP}(f, b_q, m_q, d_q) \triangleq \mathsf{spp}(f, b', m_p \mathbin{\vec{+}} m_q, d_p \mathbin{\hat{+}} d_q)$$
$$\text{where } b' = \{v \mapsto p(v) \mathbin{\vec{+}} q(v) \mid v \in \mathsf{dom}(b_p \cup b_q \cup m_p \cup m_q)\}$$
$$\mathsf{SPP}(f, b_p, m_p, d_p) \mathbin{\hat{;}} \mathsf{SPP}(f, b_q, m_q, d_q) \triangleq \mathsf{spp}(f, b', m_A \mathbin{\vec{+}} m_B, d_p \mathbin{\hat{;}} d_q)$$
$$\text{where } b' = \{v \mapsto \vec{\sum_{v' \mapsto p' \in p(v)}} \{w' \mapsto p' \mathbin{\hat{;}} q' \mid w' \mapsto q' \in q(v')\} \mid v \in \mathsf{dom}(b_p \cup b_q \cup m_p \cup m_q \cup m_A)\}$$
$$m_A = \vec{\sum_{v' \mapsto p' \in m_p}} \{w' \mapsto p' \mathbin{\hat{;}} q' \mid w' \mapsto q' \in q(v')\}, \quad m_B = \{w' \mapsto d_p \mathbin{\hat{;}} q' \mid w' \mapsto q' \in m_q\}$$
$$m_1 \mathbin{\vec{+}} m_2 \triangleq \{v \mapsto m_1(v; \bot) \mathbin{\hat{+}} m_2(v; \bot) \mid v \in \mathsf{dom}(m_1 \cup m_2)\}, \quad \vec{\Sigma} \triangleq n\text{-ary sum w.r.t. } \vec{+}$$
$$p(v) \triangleq b_p(v;\ \text{if } v \in m_p \vee d_p = \bot \text{ then } m_p \text{ else } m_p \cup \{v \mapsto d_p\}) \quad \text{(similarly for } q(v))$$

Expansion:

$p \equiv \mathsf{SPP}(f, \emptyset, \emptyset, p)$ $\quad$ if $\quad$ $p \in \{\top, \bot, \mathsf{SPP}(f', b, m, d)\}$ where $f \sqsubset f'$

Repetition:

$p^\star \triangleq \hat{\Sigma}_i\, p^i = \top \mathbin{\hat{+}} p \mathbin{\hat{+}} p \mathbin{\hat{;}} p \mathbin{\hat{+}} p \mathbin{\hat{;}} p \mathbin{\hat{;}} p \mathbin{\hat{+}} \ldots$ $\quad$ (sums in SPP converge in a finite number of steps)

Fig. 6. Definition of the SPP operations. The inductive case is identical for all operations (indicated by $\hat{+}$), except $\hat{;}$, which is given separately. Expansion inserts a trivial SPP node to reduce the remaining cases to the inductive case. The notation $b(v; d)$ means the child $b(v)$ if $v \in \mathsf{dom}(b)$, or by default d otherwise.

Operations. SPPs are closed not just under sequential composition, but under all of the NetKAT operations. These operations are largely mechanical, but more complex than for SPs, as we need to take assignments into account while respecting the conditions that ensure uniqueness. In particular, sequential composition is no longer equivalent to intersection, as the assignments in the first SPP clearly affect the tests in the second SPP. Furthermore, the star operator is no longer trivial, as the assignments in the SPP can affect the tests in the SPP itself, and a fixed point needs to be taken. We have the following operations on SPPs:

$$\hat{+}, \hat{;}, \hat{\cap}, \hat{\oplus}, \hat{\neg} \ :\ \mathsf{SPP} \times \mathsf{SPP} \to \mathsf{SPP} \qquad \hat{\star} \ :\ \mathsf{SPP} \to \mathsf{SPP} \qquad f=v, f \neq v, f \leftarrow v \ :\ \mathsf{SPP}$$

Push and pull. In addition to these operations for combining SPPs, we also have the following operations, which "push" and "pull" a symbolic packet through a SPP:

$$\mathsf{push} :\ \mathsf{SP} \times \mathsf{SPP} \to \mathsf{SP} \qquad \mathsf{pull} :\ \mathsf{SPP} \times \mathsf{SP} \to \mathsf{SP}$$

The push operation computes the effect of a SPP on a symbolic packet, giving a symbolic packet as a result. The new symbolic packet contains all of the packets that are produced by the SPP when applied to the packets in the input symbolic packet.

The pull operation simulates the effect of a SPP in reverse. This operation is used when computing the backward transition of a symbolic packet over a symbolic transition, for counter-example generation. The pull operation answers this question: given a set of output packets (represented symbolically), what are the possible input packets (also represented symbolically) that could have produced them? In other words, a concrete packet α is an element of $\mathsf{pull}(p, q)$ if and only if running the SPP p on α produces an output packet in q. In particular, it is okay if running the SPP on α produces multiple output packets, as long as at least one of them is in q.

Specifications. Each of these operations is uniquely specified by two correctness conditions:

(1) They semantically match their counterpart (e.g. $[\![p \mathbin{\hat{+}} q]\!] \equiv [\![p + q]\!]$ for all $p, q \in \text{SPP}$)
(2) They maintain the reduced & ordered invariant (e.g., $p \mathbin{\hat{+}} q$ is reduced & ordered if p, q are)

For push and pull, the correctness condition is as follows:

$$\beta \in [\![\mathsf{push}\ p\ s]\!] \iff \exists \alpha \in [\![p]\!].\ \beta \in [\![s]\!](\alpha) \quad \text{and} \quad \beta \in [\![\mathsf{pull}\ s\ p]\!] \iff \exists \alpha \in [\![p]\!].\ \alpha \in [\![s]\!](\beta)$$

For further details, see the extended version of this paper [33].

4 SYMBOLIC NETKAT AUTOMATA VIA BRZOZOWSKI DERIVATIVES

With the symbolic packet and symbolic transition representations in place, we can now define symbolic NetKAT automata. The construction of NetKAT automata shares some similarities with the construction of automata for regular expressions. In prior work [22, 37], NetKAT automata were constructed via Antimirov derivatives [2], which can be extended from regular expressions to NetKAT. The Antimirov derivative constructs a non-deterministic automaton, and as such, is well suited for handling NetKAT's union operator by inserting transitions for the two sub-terms. However, the Antimirov derivative is not well suited for our extended set of logical operators, as we cannot simply insert (non-deterministic) transitions for the operands of intersection, difference, and symmetric difference. Instead, we extend the Brzozowski derivative [13] to NetKAT, which constructs a deterministic automaton directly, and is better suited for the logical operators.

In the rest of this section, we will first describe what symbolic NetKAT automata are, and then describe how to construct them via the Brzozowski derivative.

4.1 Symbolic NetKAT Automata

An automaton for a regular expression consists of a set of states, and transitions labeled with symbols from the alphabet. Additionally, one of the states is designated as the initial state, and a subset of the states are designated as accepting states. This way, an automaton for a regular expression models the set of strings that are accepted by the regular expression.

An automaton for a NetKAT program is similar, but instead of modeling a set of strings, it models the traces that are produced for a given input packet. When a packet travels through the automaton, every state that it traverses acts as a dup operation, appending a copy of the packet to the packet's trace. Therefore, the transition between two states is labeled with a dup-free NetKAT program, represented symbolically as a SPP. That is, each edge in the automaton is not labeled with a single letter, as it would be in a standard DFA or NFA, but with a potentially large SPP. This is necessary because an edge in the automaton intuitively represents "what happens between two dups". This is precisely a dup-free NetKAT program, which can itself have rich behavior, represented canonically as an SPP. This is needed to support fine-grained control over equivalence and inclusion as discussed in Section 2—e.g., via conditionally executed dups or assignments between dups.

Secondly, instead of having a boolean at each state to determine acceptance, NetKAT automata have an additional SPP at each state that determines the set of packets that are produced as output of the automaton when a packet reaches that state. Therefore, a symbolic NetKAT automaton is a tuple $\mathcal{A} = \langle Q, q_o, \delta, \epsilon \rangle$ where Q is a finite set of states, $q_0 \in Q$ is the initial state,

- $\delta : Q \times Q \to \text{SPP}$ is the transition function, and
- $\epsilon : Q \to \text{SPP}$ is the output function.

Deterministic NetKAT automata. The notion of a deterministic NetKAT automaton is more subtle than for classic finite automata. For classic automata, a deterministic automaton is one where for every state q and symbol a, there is at most one transition from q labeled with a. For a NetKAT automaton, we need to take into account the fact that the transitions are labeled with SPPs, which

may produce multiple packets for a given input packet. Therefore, we define a deterministic NetKAT automaton as one where for every state q and packet α, the set of packets produced by $\mathsf{push}(\alpha, \delta(q, q'))$ are disjoint for all $q' \in Q$. This is equivalent to saying that $\delta(q, q'_1) \mathbin{\hat{\cap}} \delta(q, q'_2) = \bot$ for all $q'_1, q'_2 \in Q$ $(q'_1 \neq q'_2)$. Note that an input packet α may produce multiple different packets at each successor state, but these packet sets at different successor states are disjoint.

4.2 Constructing Automata via Brzozowski Derivatives

We now describe how to construct a symbolic NetKAT automaton for a NetKAT program via the Brzozowski derivative. We take the set of states to be the set of NetKAT expressions, and the initial state to be the NetKAT program itself. Because we want to construct a deterministic automaton, we need a way to represent a non-intersecting outgoing symbolic transition structure (STS). We represent such a transition structure as a NetKAT expression in the following form:

$$r \in \mathsf{STS} ::= p_1 \cdot \mathrm{dup} \cdot q_1 + \ldots + p_n \cdot \mathrm{dup} \cdot q_n$$

where $p_i \in \mathsf{SPP}$, and q_i are NetKAT expressions, and the p_i are disjoint ($p_i \mathbin{\hat{\cap}} p_j = \bot$ for $i \neq j$).

Operations. Dup is an STS, by taking $r = \top \cdot \mathrm{dup} \cdot \top$. We extend the logical operators to STSs:

$$\tilde{+}, \tilde{\cap}, \tilde{\oplus}, \tilde{-} \;:\; \mathsf{STS} \times \mathsf{STS} \to \mathsf{STS}$$

These operations need to be defined such that the resulting STS is deterministic. For $r_1 \cap r_2$:

$$(p_1 \cdot \mathrm{dup} \cdot q_1 + \ldots + p_n \cdot \mathrm{dup} \cdot q_n) \mathbin{\tilde{\cap}} (p'_1 \cdot \mathrm{dup} \cdot q'_1 + \ldots + p'_n \cdot \mathrm{dup} \cdot q'_n)$$

To bring this in STS form, we distribute the intersection over the union, and combine the terms:

$$(p_1 \mathbin{\hat{\cap}} p'_1) \cdot \mathrm{dup} \cdot (q_1 \cap q'_1) + (p_1 \mathbin{\hat{\cap}} p'_2) \cdot \mathrm{dup} \cdot (q_1 \cap q'_2) + \ldots + (p_n \mathbin{\hat{\cap}} p'_n) \cdot \mathrm{dup} \cdot (q_n \cap q'_n)$$

This is not yet in STS form, as the $q_i \cap q'_i$ terms may not all be different, so we need to collect the terms with the same $q_i \cap q'_i$, and union their SPPs. The other operators need a similar (albeit slightly more complicated) treatment in order to maintain determinism.

Second, we extend sequential composition to STSs, in two forms (denoted with the same symbol). We can compose an STS with a SPP on the left, or with a NetKAT expression on the right:

$$\tilde{\cdot} : \mathsf{SPP} \times \mathsf{STS} \to \mathsf{STS} \qquad \tilde{\cdot} : \mathsf{STS} \times \mathsf{Exp} \to \mathsf{STS}$$

Like the other operators, these need to be defined such that the resulting STS is deterministic.

All of the STS operations are defined in terms of SPP operations. Therefore, an efficient SPP implementation is crucial both for operations on symbolic packets and for operations on STSs.

Specifications. Each of these operations is uniquely specified by two correctness conditions:

(1) They semantically match their counterpart (e.g. $[\![p \mathbin{\hat{+}} q]\!] \equiv [\![p + q]\!]$ for all $p, q \in \mathsf{STS}$)
(2) They maintain the invariant that the transitions are pairwise disjoint.

Brzozowski derivative. With these operations in place, it is straightforward to define the Brzozowski derivative for NetKAT expressions:

$$\begin{array}{llll}
\epsilon(p+q) \triangleq \epsilon(p) \mathbin{\hat{+}} \epsilon(q) & \epsilon(\mathrm{dup}) \triangleq \bot & \delta(p+q) \triangleq \delta(p) \mathbin{\tilde{+}} \delta(q) & \delta(\mathrm{dup}) \triangleq \mathrm{dup} \\
\epsilon(p \cap q) \triangleq \epsilon(p) \mathbin{\hat{\cap}} \epsilon(q) & \epsilon(f{=}v) \triangleq f{=}v & \delta(p \cap q) \triangleq \delta(p) \mathbin{\tilde{\cap}} \delta(q) & \delta(f{=}v) \triangleq \bot \\
\epsilon(p \oplus q) \triangleq \epsilon(p) \mathbin{\hat{\oplus}} \epsilon(q) & \epsilon(f{\neq}v) \triangleq f{\neq}v & \delta(p \oplus q) \triangleq \delta(p) \mathbin{\tilde{\oplus}} \delta(q) & \delta(f{\neq}v) \triangleq \bot \\
\epsilon(p - q) \triangleq \epsilon(p) \mathbin{\hat{-}} \epsilon(q) & \epsilon(f{\leftarrow}v) \triangleq f{\leftarrow}v & \delta(p - q) \triangleq \delta(p) \mathbin{\tilde{-}} \delta(q) & \delta(f{\leftarrow}v) \triangleq \bot \\
\epsilon(p \cdot q) \triangleq \epsilon(p) \mathbin{\hat{\cdot}} \epsilon(q) & \epsilon(\top) \triangleq \top & \delta(p \cdot q) \triangleq \delta(p) \mathbin{\tilde{\cdot}} q \mathbin{\tilde{+}} \epsilon(p) \mathbin{\tilde{\cdot}} \delta(q) & \delta(\top) \triangleq \bot \\
\epsilon(p^\star) \triangleq \epsilon(p)^\star & \epsilon(\bot) \triangleq \bot & \delta(p^\star) \triangleq \epsilon(p)^\star \mathbin{\tilde{\cdot}} \delta(p) \mathbin{\tilde{\cdot}} p^\star & \delta(\bot) \triangleq \bot
\end{array}$$

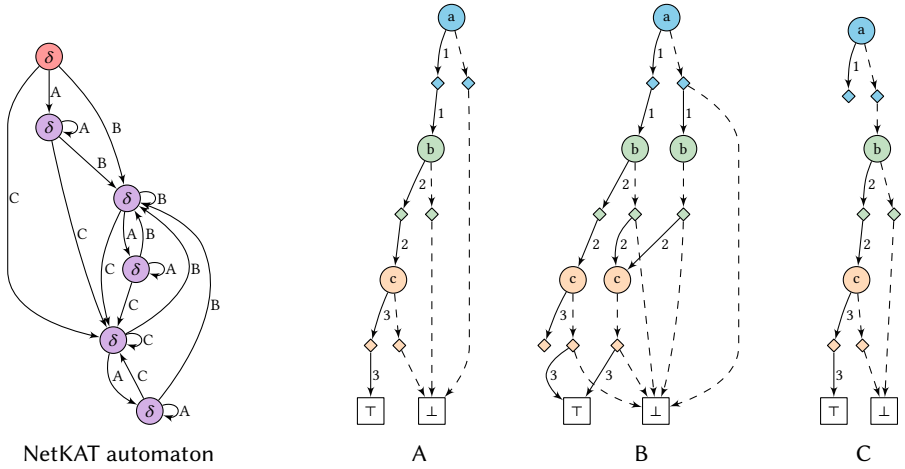

Fig. 7. Symbolic NetKAT automaton for $p \triangleq \left((a \leftarrow 1 \cdot b \leftarrow 2 \cdot c \leftarrow 3 \cdot \mathsf{dup})^\star + (b{=}2 \cdot c{=}3 \cdot \mathsf{dup})^\star\right)^\star$

We construct a deterministic symbolic NetKAT automaton for a NetKAT program p as follows:

States $Q \triangleq \mathsf{Exp}$.
Initial state $q_0 \triangleq p$.
Transitions take $\delta(q, q')$ to be the SPP of q' in the Brzozowski derivative of $\delta(q)$.
Output $\epsilon : Q \to \mathsf{SPP}$, defined above.

The Brzozowski derivative is guaranteed to transitively reach only finitely many essentially different NetKAT expressions from a given start state. Therefore, in the actual implementation, we do not use the infinite set Exp for the states, but instead use only the finitely many essentially different NetKAT terms reached from the start state.

The NetKAT automaton for an example program is shown in Figure 7. The edges are labeled with the SPPs that represent the transitions. The output SPP of every state is $\top$, i.e., the automaton accepts all packets that reach a state. Note that the SPPs in the figure feature several diamonds without outgoing edges. These diamonds produce no output packet, i.e., the packet is dropped.

For more details, see the extended version of this paper [33].

5 BISIMILARITY AND COUNTER-EXAMPLE GENERATION

After converting the NetKAT programs to automata, we can check equivalence of two NetKAT programs by checking whether their automata are bisimilar. Bisimilarity is a well-known notion of equivalence for automata. For NetKAT automata, two states are bisimilar for a given packet α if (1) the states produce the same output packets, and (2) for each possible modified packet α', the two states transition to states that are bisimilar for α'.

Historically, methods for computing bisimulations of automata [8, 18] have been based on either on Moore's algorithm for minimization [35] or Hopcroft and Karp's algorithm [25]. Indeed, the naive approach shown in Figure 2 follows the basic structure of Hopcroft-Karp, in the sense that it relates the start states and proceeds to follow transitions forward in the two automata.

Our situation is different, because we have already added the symmetric difference operator to NetKAT. Therefore, we can reduce equivalence checks $A \equiv B$ to emptiness checks $A \oplus B \equiv \bot$.

Subtlety of symmetric difference. The reader should note that the situation is a bit more subtle than it may seem at first sight. In particular, consider the query $A \equiv B$, which asks whether two

$done(q) \leftarrow \bot$ for $q \in Q$;
$todo(q) \leftarrow \bot$ for $q \in Q \setminus \{q_0\}$;
$todo(q_0) \leftarrow \top$;
while $\exists q.\ todo(q) \neq \bot$ **do**
  set $p := todo(q) \mathbin{\hat{\cdot}} done(q)$;
  $todo(q) \leftarrow \bot$;
  $done(q) \mathbin{\hat{+}}= p$;
  for $q' \in Q$ **do**
    $todo(q') \mathbin{\hat{+}}= \mathsf{push}(p, \delta(q, q'))$;
return $\sum_{q \in Q} \mathsf{push}(done(q), \epsilon(q))$;

(i) Forward algorithm

$done(q) \leftarrow \bot$ for $q \in Q$;
$todo(q) \leftarrow \mathsf{pull}(\epsilon(q), \top)$ for $q \in Q$;

while $\exists q.\ todo(q) \neq \bot$ **do**
  set $p := todo(q) \mathbin{\hat{\cdot}} done(q)$;
  $todo(q) \leftarrow \bot$;
  $done(q) \mathbin{\hat{+}}= p$;
  for $q' \in Q$ **do**
    $todo(q') \mathbin{\hat{+}}= \mathsf{pull}(\delta(q', q), p)$;
return $done(q_0)$;

(ii) Backward algorithm

Fig. 8. Forward and backward algorithms for NetKAT automata

NetKAT programs A and B are equivalent. If they are inequivalent, then there must be some input packet that causes A to produce a different output than B. Output in this sense does not just mean the final output packet, but also the trace of the packet. It can be the case that A and B produce the same final output packets, but differ in the trace of the packets. Therefore, the symmetric difference $A \oplus B$ takes the symmetric difference of the traces, not just of the output packets. In other words, the set $[\![A \oplus B]\!]$ is the set of counter-example traces to the equivalence of A and B.

Subtlety of the emptiness of an automaton. A second subtlety is the emptiness of an automaton. Whereas for regular expressions, it is easy to check whether a DFA is empty (just check whether there are any reachable accepting states), this is not the case for NetKAT automata. Because NetKAT automata manipulate and test the fields of packets, it is possible that a packet travels through the automaton, and even causes multiple packets to be produced (e.g., due to the presence of union in the original NetKAT expression), but nevertheless, it is possible that all of these packets are eventually dropped by the automaton, before producing any output packets.

The goal of this section is to develop a symbolic algorithm for this check, which is more efficient than the naive algorithm that checks all concrete input packets separately.

5.1 Forward Algorithm

To check whether a NetKAT automaton drops all input packets, we use an algorithm that works in the forward direction. The forward algorithm computes *all* output packets that the automaton can produce, across all possible input packets. More formally: given a NetKAT automaton for a NetKAT program p, the forward algorithm computes the set of final packets occurring in the traces $[\![p]\!](\alpha)$ across all input packets α. The forward algorithm therefore starts with the complete symbolic packet $\top$ at the input state, and repeatedly applies all outgoing transitions δ to it. In this way, the algorithm iteratively accumulates a symbolic packet at every state, which represents the set of packets that can reach that state from the start state. Once we know the set of packets that can reach a state, we can determine the set of output packets by applying the output function ϵ to the symbolic packet of each state, and taking the union of the results.

The forward algorithm is shown in Figure 8. To determine the set of packets that a given NetKAT expression p can produce, we convert it to a NetKAT automaton, and then use the forward algorithm to determine the symbolic output packet. In our implementation, we also have a way to stop the algorithm early, if the user is only interested in a "yes" or "no" answer for the query $p \equiv \bot$. In this case, we can stop the algorithm as soon as any output packet is produced.

5.2 Backward Algorithm

The forward algorithm gives us the set of output packets that a NetKAT automaton can produce, but we are also interested in which input packets cause this output to be produced. For instance, for a given check $A \equiv B$, we want to know *which* input packets cause A and B to produce different sets of output traces. More generally, we want to know which input packets cause a NetKAT automaton to produce a nonempty set of output packets or, formally, given an automaton for a NetKAT program p, what is the set of initial packets α for which $[\![p]\!](\alpha)$ is nonempty.

To answer this question, we developed a *backward algorithm*, shown in Figure 8. The backward algorithm computes *all* input packets that can cause the automaton to produce a nonempty set of output packets. The backward algorithm therefore starts with the complete symbolic packet $\top$ at every state, and pulls it backwards through all observation functions ϵ. The algorithm then iteratively accumulates a symbolic packet at every state, which represents the set of packets that will cause the automaton to produce a nonempty set of output packets, when starting from that state. The accumulation is done by pulling the symbolic packet backwards through all transitions until a fixpoint is reached. Once the fixpoint is reached, we return the symbolic packet at the start state, which represents the set of input packets that will cause the automaton to produce a nonempty set of output packets when starting from the start state.

6 IMPLEMENTATION

We have implemented the algorithms in a new system, KATch, comprising 2500 lines of Scala. In this section we explain the system's interface. The implementation provides a surface syntax for expressing queries, which extends the core NetKAT syntax from Figure 1. The extended syntax is shown in Figure 9. The language has statements and expressions, which we describe below.

Statements

check $e_1 \equiv e_2$
check $e_1 \not\equiv e_2$
print e
$x = e$
for $i \in n_1..n_2$ do c

Expressions

forward e, backward e
$e_1 \cap e_2$, $e_1 \oplus e_2$, $e_1 - e_2$
$\hat{\exists}\, f\ e$, $\hat{\forall}\, f\ e$
(+ NetKAT, see fig. 1)

Fig. 9. NKPL syntax.

Statements. The check $e_1 \equiv e_2$ command runs the bisimulation algorithm. The system reports success if the expressions are equivalent (when $\equiv$ is used) or inequivalent (when $\not\equiv$) is used, or reports failure otherwise. The other statements behave as expected: The print statement invokes the pretty printer, $x = e$ let-binds names to expressions, and for runs a statement in a loop.

Expressions. The forward e expression computes, in forward-flowing mode, the set of output packets resulting from running the symbolic packet $\top$ through the given expression e (see Figure 8). Conversely backward e computes the set of input packets which generate some output packet when run on the given expression (Figure 8). These are generally used in conjunction with the $\oplus$, $-$, $\cap$ operators to express the desired query. The user may combine these expressions with the $\hat{\exists}$ and $\hat{\forall}$ operators to reason about symbolic packets, and the print operator to pretty print the symbolic packets, or the check operator to assert (in)equivalence of two expressions.

We note that the generality of the language allows us to express some queries in different, equivalent ways. For example, the two checks: $e_1 \equiv e_2$ and $e_1 \oplus e_2 \equiv \bot$ are equivalent. However, the expression on the right lends itself to inspection of counterexample input packets: print (backward $e_1 \oplus e_2$). This statement pretty prints a symbolic packet having different behavior on e_1 and e_2 (i.e., packets that result in some valid history in one expression and not the other). In other words, it prints the set of all counter-example input packets for $e_1 \equiv e_2$.

Topologies and routing tables. While NetKAT can be used to specify routing policies declaratively, it is also possible to import topologies and routing tables into NetKAT. For example, a simple way to define routing tables and topologies in NetKAT is as follows:

$$\begin{array}{c} \mathsf{R} \triangleq \mathsf{sw}{=}5 \cdot (\mathsf{dst}{=}6 \cdot \mathsf{pt}{\leftarrow}3 + \mathsf{dst}{=}8 \cdot \mathsf{pt}{\leftarrow}4) \\ + \cdots + \\ \mathsf{sw}{=}7 \cdot (\mathsf{dst}{=}8 \cdot \mathsf{pt}{\leftarrow}2 + \mathsf{dst}{=}9 \cdot \mathsf{pt}{\leftarrow}1) \end{array} \quad \left| \quad \begin{array}{c} \mathsf{T} \triangleq \mathsf{sw}{=}5 \cdot (\mathsf{pt}{=}3 \cdot \mathsf{sw}{\leftarrow}6 + \mathsf{pt}{=}4 \cdot \mathsf{sw}{\leftarrow}8) \\ + \cdots + \\ \mathsf{sw}{=}7 \cdot (\mathsf{pt}{=}2 \cdot \mathsf{sw}{\leftarrow}8 + \mathsf{pt}{=}1 \cdot \mathsf{sw}{\leftarrow}9) \end{array} \right.$$

The routing R tests the switch field (where the packet currently is), and the destination field (where the packet is supposed to end up), and then sets the port over which the packet should be sent out. The topology T tests the switch and port fields, and then transports the packet to the next switch. We define the action of the network by composing the route and topology:

$$\mathsf{net} \triangleq \mathsf{R} \cdot \mathsf{T} \cdot \mathsf{dup}$$

We include a dup to extend the trace of the packet at every hop.

Example 6.1 (All-Pairs Reachability Queries). A naive way to check reachability of all pairs of hosts in a network is to run the following command for each pair of end hosts n_i, n_j:

$$\mathsf{check}\ (\mathsf{sw}{=}n_i) \cdot \mathsf{net}^\star \cdot (\mathsf{sw}{=}n_j) \not\equiv \bot$$

Of course, this requires a number of queries which is quadratic in the number of hosts—quickly becoming prohibitive. One might think that we could reduce the number of queries to n by running:

$$\text{for } i \in 1..n \text{ do } \mathsf{check}\ (\mathsf{forward}\ (\mathsf{sw}{=}i \cdot \mathsf{net}^\star)) \equiv (\mathsf{sw} \in 1..n)$$

This query does work if sw is the only field of our packets, because the left hand side contains the packets that can be reached from i via the network, and the right hand side contains packets where the sw field is any value in $1..n$. Unfortunately, this does not quite work if the network also operates on other packet fields (say, fields f_1 and f_2), as the packets on the left hand side will have those fields, whereas the packets on the right hand side will only have a sw field. The $\hat{\exists}$ and $\hat{\forall}$ operators allow us to manipulate symbolic packets and express all pairs reachability in n queries:

$$\text{for } i \in 1..n \text{ do } \mathsf{check}\ (\hat{\exists} f_1\ (\hat{\exists} f_2\ (\mathsf{forward}\ (\mathsf{sw} = i \cdot \mathsf{net}^\star)))) \equiv (\mathsf{sw} \in 1..n)$$

The operators $\hat{\exists}$ and $\hat{\forall}$ give the programmer the ability to reason about symbolic packets that may be computed, for instance, by forward or backward. The specifications are:

$$p \in \hat{\exists} f\ e \overset{\triangle}{\iff} \exists v \in V : p[f{\leftarrow}v] \in [\![e]\!] \quad \text{and} \quad p \in \hat{\forall} f\ e \overset{\triangle}{\iff} \forall v \in V : p[f{\leftarrow}v] \in [\![e]\!]$$

The implementation does not iterate over V (indeed, V can be unknown). Rather, $\hat{\exists}$ and $\hat{\forall}$ are implemented directly as operations on symbolic packets. In fact, the implementation does not need to fix the sets of fields or values up-front at all. Instead, it operates on a conceptually infinite set of fields and values. This works because SPPs' default cases handle all remaining fields and all values.

Correctness and testing methodology. We validated our implementation with the following property-based fuzz testing methodology:

(1) We implemented the semantics $[\![p]\!](\alpha)$ in Scala, as described in Figure 1.
(2) We repeatedly pick two SPs/SPPs p and q and a packet α. We enumerated small SPPs up to a bound and generated larger SPPs randomly. We generated packets exhaustively.
(3) We check the soundness (e.g., $[\![p \mathbin{\hat{+}} q]\!](\alpha) = [\![p]\!](\alpha) \cup [\![q]\!](\alpha)$) and canonicity (e.g., $[\![p]\!](\alpha) = [\![q]\!](\alpha)$ for all α if and only if $p = q$) of the operations, as well as additional algebraic laws derived from the NetKAT axioms (e.g., $p \mathbin{\hat{+}} q = q \mathbin{\hat{+}} p$).

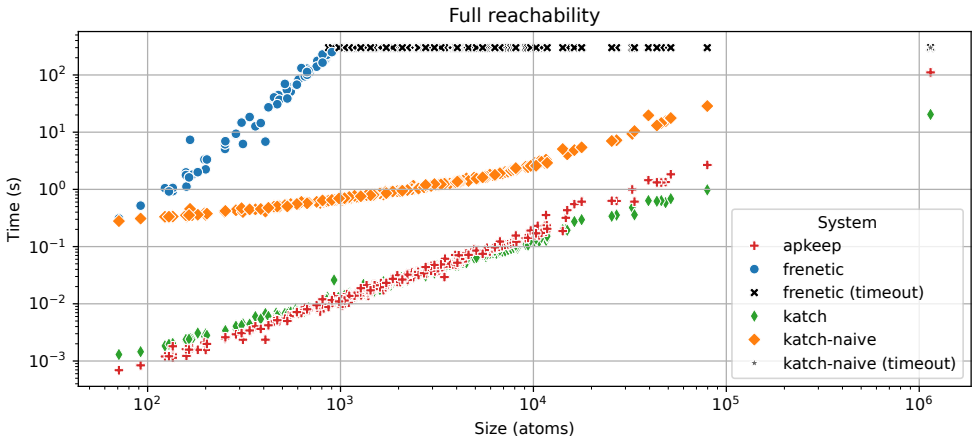

Fig. 10. Full reachability queries on Topology Zoo. KATch and APKeep results are averages of 100 runs, preceded by 10 runs of JIT warmup. KATch-naive is without JIT warmup and uses $O(n^2)$ 1-to-1 queries.

In addition to property-based testing for SPs and SPPs, we also generated hundreds of thousands of pairs of random NetKAT expressions and checked that KATch's bisimilarity check matches the output of Frenetic. This revealed a subtle bug in Frenetic, which we reported and is now fixed.

7 EVALUATION

To evaluate KATch, we conducted experiments in which we used it to solve a variety of verification tasks for a range of topologies and routes, as well as challenging combinatorial NetKAT terms. The goal of our evaluation is to answer the following three questions:

(1) How does KATch perform compared to the state of the art NetKAT verifier, Frenetic?
(2) How does KATch perform compared to the state of the art specialized network verification tool, APKeep?
(3) How well does KATch scale with the size of topology?
(4) When does KATch perform asymptotically better than prior work?

7.1 Topology Zoo

To begin to answer the first three questions, we conducted our experiments using *The Internet Topology Zoo* [28] dataset, a publicly available set of 261 network topologies, ranging in size from just 4 nodes (the original ARPANet) to 754 nodes (KDL, the Kentucky Data Link ISP topology). For each topology, we generated a destination-based routing policy using an all-pairs shortest path scheme that connects every pair of routers to each other.

To demonstrate KATch's scalability, we first ran full (i.e., $O(n^2)$) reachability queries for every topology in the zoo using KATch, Frenetic[2], and APKeep [47]. The results are shown in Figure 10 in a log-log plot, so straight lines correspond to polynomials with exponents related to their slope.

The figure shows two different configurations of KATch: one that verifies full reachability using a quadratic number of point-to-point queries and does not use JIT warmup ("katch-naive"), and one that verifies full reachability using a linear number of queries and does use JIT warmup ("katch").

Because of the size of the dataset, we set a timeout of 5 minutes per topology. Under these conditions, Frenetic was unable to complete for all but the smallest topologies. KATch-naive

[2]https://github.com/frenetic-lang/frenetic

Name (Topology Zoo)	Size (atoms)	1-to-1 reachability KATch	1-to-1 reachability Frenetic	Slicing KATch	Slicing Frenetic	Full reachability KATch	Full reachability APKEEP
Layer42	135	0.00	0.04	0.00	0.07	0.00	0.00
Compuserv	539	0.00	0.35	0.01	0.81	0.01	0.01
Airtel	785	0.01	0.82	0.02	1.98	0.01	0.01
Belnet	1388	0.01	3.19	0.03	7.99	0.02	0.01
Shentel	1865	0.01	3.93	0.03	9.76	0.02	0.03
Arpa	1964	0.01	4.23	0.04	10.42	0.03	0.03
Sanet	4100	0.03	23.42	0.07	62.21	0.05	0.07
Uunet	5456	0.04	80.85	0.11	203.80	0.07	0.07
Missouri	9680	0.06	166.87	0.21	441.72	0.13	0.19
Telcove	10720	0.07	441.47	0.21	1121.30	0.12	0.16
Deltacom	27092	0.18	2087.34	0.48	5098.57	0.36	0.63
Cogentco	79682	0.53	18910.82	1.38	54247.73	0.98	2.67
Kdl	1144691	9.87	out of memory	23.67	out of memory	19.44	110.92

Fig. 11. Running time (in seconds) of KATch, FRENETIC, and APKEEP on Topology Zoo queries. KATch and APKEEP results are averages of 100 runs, preceded by 10 runs of JIT warmup.

handles most of the topologies in well under a second, and all but the largest in under 2 minutes. KATch-naive exceeds the timeout on Kentucky Data Link—using a quadratic number of queries to check full reachability produces over 500k individual point-to-point queries in a network with 754 nodes! (However, KATch-naive is able to finish it in just under 20 minutes.)

To avoid combinatorial blowup in the verification query itself, we also used KATch's high-level verification interface, to check full reachability using a *linear* number of queries, as discussed in Example 6.1. FRENETIC does not have an analogous linear mode. APKEEP does have an analogous mode, as it has specialized support for full reachability queries. APKEEP is faster than KATch for the smaller topologies, but slower for the larger ones. Indeed, the slope of the APKEEP line is steeper than the KATch line, indicating that KATch scales slightly better with the size of the network on these queries. It is important to note that APKEEP has support for prefix-matching, ACLs, NAT, incrementality, and other features for which KATch does not have specialized support and which are not tested in this comparison. One should therefore not draw strong conclusions from this comparison; we include it as it is encouraging that reachability via NetKAT equivalence can be competitive with a state-of-the-art specialized tool.

We also randomly sampled a subset of toplogies from the Topology Zoo of varying size and generated point-to-point (1-to-1) reachability and slicing queries to be checked against the routing configurations with no timeout. For each query, we ran KATch and also generated an equivalent query in the syntax of FRENETIC, and ran FRENETIC's bisimulation verifier on those queries. We present a full table of results of these experiments in Figure 11. The table shows that KATch's relative speedup over FRENETIC is considerable, and increases problem size. This is encouraging, because it shows that KATch is more scalable. FRENETIC was unable to complete KDL within a 200GB memory limit (for comparison, KATch is able to complete the same query with well under 1GB). We also include a selection of full-reachability results from Figure 10 in the table. The reader may wonder why full reachability takes only twice as long as 1-to-1 reachability. This is because a significant fraction of the time is spent constructing automata (which are similar for both queries), and much of the work in the full reachability queries is shared due to memoization of SPP operations.

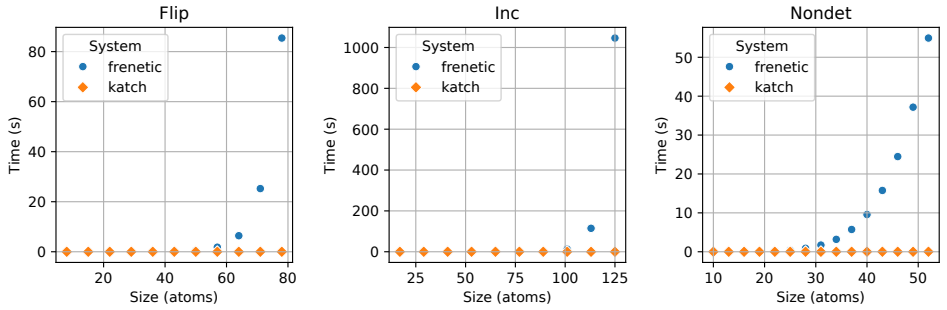

Fig. 12. Results of running KATch and Frenetic on combinatorial benchmarks

7.2 Combinatorial Examples

Finally, we ran experiments to test the hypothesis that SPPs have an asymptotic advantage for certain types of queries. Frenetic uses forwarding decision diagrams (FDDs), which is a different (and non-canonical) representation for automaton transitions. A key advantage of SPPs over FDDs is that SPPs keep the updates to each field next to the tests of the same field. On the other hand, FDDs keep all updates at the leaves, which can result in combinatorial explosion during sequencing. To test this, we generated the following NetKAT programs:

Inc: Treating the input packet's n boolean fields as a binary number, increment it by one.
Flip: Sequentially flip the value of each of the n boolean fields.
Nondet: Set each field of the packet to a range of values from 0 to n.

For Inc, we tested that repeatedly incrementing (using the $\star$ operator) can turn packet $00\cdots0$ into $11\cdots1$. For Flip, we tested that flipping all bits twice returns the original packet. For Nondet, we tested that setting the fields non-deterministically twice is the same as doing it once. The results of this experiment are shown in Figure 12. Because the available fields are hardcoded in Frenetic, we only ran the Inc and Flip experiments up to $n = 10$. We ran the non-determinism test up to $n = 15$. KATch finishes all three queries up to $n = 100$ in under a one minute, demonstrating its asymptotic advantage for these queries, while Frenetic shows combinatorial blowup on larger packets.

Source of speedup. The speedup of KATch over Frenetic comes from several sources: (1) the use of SPPs instead of FDDs, which can be exponentially more compact and support more efficient sequential composition, (2) the symbolic bisimulation algorithm operating on SPs, which can handle exponentially large sets of packets at once, and (3) a quadratic-doubling implementation of star, which can handle exponentially long traces quickly. These differences show up most strongly in the combinatorially adversarial examples above, but the speedup is also considerable for the Topology Zoo benchmarks, which are based on real-world topologies and not designed to be adversarial.

Field order. The field order of SPs and SPPs can affect performance, just as for BDDs. In practice, field orders that keep related fields close together are beneficial. Our implementation allows the user to control the field order, but we did not use this for our benchmarks. By default, the fields are in the order in which they first occur in the input file, which turned out to work well enough.

8 RELATED WORK

This section discusses the most closely related prior work to this paper, focusing on three areas: NetKAT, network verification, and automata-theoretic approaches to verification.

NetKAT. NetKAT was originally proposed as a semantic foundation for SDN data planes [1]. Indeed, being based on KAT [29], the language provides a sound and complete algebraic reasoning system. Later work on NetKAT developed an automata-theoretic (or coalgebraic) account of the language [22], including a decision procedure based on Brzozowski derivatives and bisimulation, implemented in Frenetic. However, the performance of this approach turns out to be poor, as shown in our experiments, due to the use of ad hoc data structures ("bases") to encode packets and automata. NetKAT's compiler uses a variant of BDDs, called Forwarding Decision Diagrams (FDDs), as well as an algorithm for converting programs to automata using Antimirov derivatives [37], improving on the earlier representation of transitions as sets of bases [22]. The SPPs proposed in this paper improve on FDDs by ensuring uniqueness and supporting efficient sequential composition in the common case. In addition, the deterministic automata used in KATch support additional "negative" operators that are useful for verification. Other papers based on NetKAT have explored use of the language in other settings such as distributed control planes [7], probabilistic networks [21, 39, 40], and Kleene algebra over composable theories with unbounded state [23]. It would be interesting to extend the techniques developed in this paper to these richer settings. Another interesting direction for future work is to build a symbolic verifier for the guarded fragment of NetKAT [38].

Network Verification. Early work by Xie et al. [45] proposed a unifying mathematical model for Internet routers and developed algorithms for analyzing network-wide reachability. Although the paper did not discuss an implementation, its elegant formal model has been extremely influential in the community and has served as the foundation for many follow-on efforts, including this work. The emergence of software-defined networking (SDN) led to a surge of interest in static data plane verification, including systems such as Header Space Analysis (HSA) [26], Anteater [32], VeriFlow [27], Atomic Predicates (AP) [46], APKeep [47]. These systems all follow a common approach: they build a model of the network-wide forwarding behavior and then check whether given properties hold. However, they vary in the data structures and algorithms used to represent and analyze the network model. For instance, Anteater relies on first-order logic and SAT solvers, while VeriFlow uses prefix trees and custom graph-based algorithms. HSA, AP and APKeep are arguably the most related to our work as they use symbolic representations and BDDs respectively.

The primary difference between KATch and these systems is that the latter implement specialized algorithms for network-wide analysis queries, while KATch (and Frenetic) solve the more general NetKAT equivalence problem, into which network-wide analysis queries can be encoded. This generic approach is based on principled automata-theoretic foundations, but one might expect it to be less efficient than specialized algorithms. Frenetic was found to be comparable to or faster than HSA [22], so by transitivity we expect KATch to be faster than HSA. However, the current state of the art is APKeep, which is significantly faster than Frenetic and HSA. We include a comparison with APKeep in Section 7 on reachability queries. The results show that KATch is competitive with APKeep on these benchmarks, despite solving general NetKAT queries. As discussed, one should not draw strong conclusions from this comparison as APKeep includes additional features; nevertheless, it offers a promising preliminary indication of NetKAT's scalability and performance.

Another line of work has explored how to lift verification from the data plane to the control plane. Batfish [10, 19] proposed using symbolic simulation to analyze distributed control planes—i.e., generating all possible data planes that might be produced starting from a given control-plane configuration. Like KATch, Batfish uses a BDD-based representation for data plane analysis. MineSweeper [4] improves on Batfish using an SMT encoding of the converged states of the control plane that avoids having to explicitly simulate the underlying routing protocols. Recent work has focused on using techniques like modular reasoning [41, 42] abstract interpretation [6], and a form of symmetry reduction [5] to further improve the scalability of control-plane verification.

Automata-Theoretic Approach and Symbolic Automata. Our work on KATch builds on the large body of work on the automata-theoretic approach to verification and symbolic automata. The automata-theoretic approach was pioneered in the 1980s, with applications of temporal logics and model checking to hardware verification [44]. BDDs, originally proposed by Lee [30], were further developed by Bryant [11], and used by McMillan for symbolic model checking [14]. BDD-based techniques were a success story of symbolic model checking of hardware in the 1990s—Chaki and Gurfinkel give an overview [15].

An influential line of work by D'Antoni and Veanus developed techniques for representing and transforming finite automata where the transitions are not labeled with individual characters but with elements of a so-called effective Boolean algebra [16, 17]. Shifting from a concrete to a symbolic representation requires generalizing classical algorithms, such as minimization and equivalence, but facilitates building automata that work over huge alphabets, such as Unicode.

Pous [36] developed symbolic techniques for checking the equivalence of automata where transitions are specified using BDDs. The methods developed in this paper take Pous's work as a starting point but develop a non-trivial extension for NetKAT. In particular, SPPs provide a compact representation for "carry-on" packets, which is a critical and unique aspect of NetKAT's semantics. Bonchi and Pous [8] explored the use of up-to techniques for checking equivalence of automata. In principle, one can view the invariants enforced by KATch's term representations as a kind of up-to technique, but a full investigation of this idea requires more work.

Our backward algorithm for computing bisimulations can be seen as a variant of Moore's classic algorithm for computing the greatest bisimulation of classic automata. Doenges et al. [18] proposed an analogous approach for checking equivalence of automata that model the behavior of P4 packet parsers [9]. However, Leapfrog's model is simpler than NetKAT's and is based on classic finite automata. It achieves scalability due to a novel up-to technique that "leaps" over internal buffering transitions rather than symbolic representations.

ACKNOWLEDGEMENTS

We wish to thank the PLDI reviewers for many helpful comments and suggestions, which improved the paper significantly. We are also grateful to Aaron Gember-Jacobson and Peng Zhang for making AP Keep available as open-source software, to Caleb Koch for early discussions on compact data structures for data plane verification, to the Cornell PLDG for helpful comments on early drafts, and to the EPFL DCSL group for providing a welcoming and supportive environment. This work was supported in part by ONR grant N68335-22-C-0411, DARPA grant W912CG-23-C-0032, ERC grant 101002697, and a Royal Society Wolfson fellowship.

DATA AVAILABILITY

The current version of the implementation is available at https://github.com/cornell-netlab/KATch. A snapshot of the code as of artifact evaluation (with a dependencies-included Docker image) is available on Zenodo [34].

REFERENCES

[1] Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. 2014. NetKAT: Semantic Foundations for Networks. In *POPL*. https://doi.org/10.1145/2535838.2535862

[2] Valentin Antimirov. 1996. Partial derivatives of regular expressions and finite automaton constructions. *Theoretical Computer Science* 155, 2 (Mar 1996), 291–319. https://doi.org/10.1016/0304-3975(95)00182-4

[3] Michael Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. 2005. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In *FMCO*. https://doi.org/10.1007/11804192_17

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

[5] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. 2018. Control Plane Compression. In *SIGCOMM*. https://doi.org/10.1145/3230543.3230583

[6] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. 2020. Abstract Interpretation of Distributed Network Control Planes. In *POPL*. https://doi.org/10.1145/3371110

[7] Ryan Beckett, Ratul Mahajan, Todd Millstein, Jitendra Padhye, and David Walker. 2016. Don't Mind the Gap: Bridging Network-Wide Objectives and Device-Level Configurations. In *SIGCOMM*. https://doi.org/10.1145/2934872.2934909

[8] Filippo Bonchi and Damien Pous. 2013. Checking NFA Equivalence with Bisimulations up to Congruence. In *POPL*. https://doi.org/10.1145/2429069.2429124

[9] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. 2014. P4: Programming Protocol-Independent Packet Processors. *SIGCOMM Computer Communication Review* 44, 3 (Jul 2014), 87–95. https://doi.org/10.1145/2656877.2656890

[10] Matt Brown, Ari Fogel, Daniel Halperin, Victor Heorhiadi, Ratul Mahajan, and Todd D. Millstein. 2023. Lessons from the evolution of the Batfish configuration analysis tool. In *SIGCOMM*. https://doi.org/10.1145/3603269.3604866

[11] Randal E. Bryant. 1986. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Trans. Comput.* C-35, 8 (Aug 1986), 677–691. https://doi.org/10.1109/TC.1986.1676819

[12] Randal E. Bryant. 1992. Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams. *Comput. Surveys* 24, 3 (Sep 1992), 293–318. https://doi.org/10.1145/136035.136043

[13] Janusz A. Brzozowski. 1962. Canonical regular expressions and minimal state graphs for definite events. In *Proceedings of the Symposium of Mathematical Theory of Automata*.

[14] Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and L. J. Hwang. 1990. Symbolic Model Checking: 10^20 States and Beyond. In *LICS*. https://doi.org/10.1109/LICS.1990.113767

[15] Sagar Chaki and Arie Gurfinkel. 2018. BDD-Based Symbolic Model Checking. In *Handbook of Model Checking*, Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem (Eds.). Springer International Publishing, 219–245. https://doi.org/10.1007/978-3-319-10575-8_8

[16] Loris D'Antoni and Margus Veanes. 2014. Minimization of Symbolic Automata. In *POPL*. https://doi.org/10.1145/2535838.2535849

[17] Loris D'Antoni and Margus Veanes. 2017. Forward Bisimulations for Nondeterministic Symbolic Finite Automata. In *TACAS*. https://doi.org/10.1007/978-3-662-54577-5_30

[18] Ryan Doenges, Tobias Kappé, John Sarracino, Nate Foster, and Greg Morrisett. 2022. Leapfrog: Certified Equivalence for Protocol Parsers. In *PLDI*. https://doi.org/10.1145/3519939.3523715

[19] Ari Fogel, Stanley Fung, Luis Pedrosa, Meg Walraed-Sullivan, Ramesh Govindan, Ratul Mahajan, and Todd Millstein. 2015. A General Approach to Network Configuration Analysis. In *NSDI*.

[20] Nate Foster, Rob Harrison, Michael J. Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story, and David Walker. 2011. Frenetic: A Network Programming Language. In *ICFP*. https://doi.org/10.1145/2034773.2034812

[21] Nate Foster, Dexter Kozen, Konstantinos Mamouras, Mark Reitblatt, and Alexandra Silva. 2016. Probabilistic NetKAT. In *ESOP*. https://doi.org/10.1007/978-3-662-49498-1_12

[22] Nate Foster, Dexter Kozen, Mae Milano, Alexandra Silva, and Laure Thompson. 2015. A Coalgebraic Decision Procedure for NetKAT. In *POPL*. https://doi.org/10.1145/2676726.2677011

[23] Michael Greenberg, Ryan Beckett, and Eric Campbell. 2022. Kleene Algebra Modulo Theories: A Framework for Concrete KATs. In *PLDI*. https://doi.org/10.1145/3519939.3523722

[24] Pieter Hooimeijer, Benjamin Livshits, David Molnar, Prateek Saxena, and Margus Veanes. 2011. Fast and Precise Sanitizer Analysis with BEK. In *USENIX Conference on Security*.

[25] John E. Hopcroft and Richard M. Karp. 1971. A Linear Algorithm for Testing Equivalence of Finite Automata.

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

[27] Ahmed Khurshid, Wenxuan Zhou, Matthew Caesar, and P. Brighten Godfrey. 2012. Veriflow: Verifying Network-Wide Invariants in Real Time. *SIGCOMM Computer Communications Review* 42, 4 (Sep 2012), 467–472. https://doi.org/10.1145/2377677.2377766

[28] Simon Knight, Hung X. Nguyen, Nickolas Falkner, Rhys Bowden, and Matthew Roughan. 2011. The Internet Topology Zoo. *IEEE Journal on Selected Areas in Communications* 29, 9 (Oct 2011), 1765 –1775. https://doi.org/10.1109/JSAC.2011.111002

[29] Dexter Kozen. 1996. Kleene Algebra with Tests and Commutativity Conditions. In *TACAS*. https://doi.org/10.1007/3-540-61042-1_35

[30] C. Y. Lee. 1959. Representation of switching circuits by binary-decision programs. *The Bell System Technical Journal* 38, 4 (Jul 1959), 985–999. https://doi.org/10.1002/j.1538-7305.1959.tb01585.x

[31] K. Rustan M. Leino and Valentin Wüstholz. 2014. The Dafny Integrated Development Environment. In *F-IDE*. https://doi.org/10.4204/EPTCS.149.2

[32] Haohui Mai, Ahmed Khurshid, Rachit Agarwal, Matthew Caesar, P. Brighten Godfrey, and Samuel Talmadge King. 2011. Debugging the Data Plane with Anteater. *SIGCOMM Computer Communications Review* 41, 4 (Aug 2011), 290–301. https://doi.org/10.1145/2043164.2018470

[33] Mark Moeller, Jules Jacobs, Olivier Savary Belanger, David Darais, Cole Schlesinger, Steffen Smolka, Nate Foster, and Alexandra Silva. 2024. KATch: A Fast Symbolic Verifier for NetKAT. arXiv:2404.04760 [cs.PL] https://arxiv.org/pdf/2404.04760.pdf

[34] Mark Moeller, Jules Jacobs, Olivier Savary Belanger, David Darais, Cole Schlesinger, Steffen Smolka, Nate Foster, and Alexandra Silva. 2024. *KATch: A Fast Symbolic Verifier for NetKAT*. https://doi.org/10.5281/zenodo.10961123

[35] Edward F. Moore. 1956. Gedanken-Experiments on Sequential Machines. In *Automata Studies. (AM-34), Volume 34*. https://doi.org/10.1515/9781400882618-006

[36] Damien Pous. 2015. Symbolic Algorithms for Language Equivalence and Kleene Algebra with Tests. In *POPL*. https://doi.org/10.1145/2775051.2677007

[37] Steffen Smolka, Spiridon Eliopoulos, Nate Foster, and Arjun Guha. 2015. A Fast Compiler for NetKAT. In *ICFP*. https://doi.org/10.1145/2784731.2784761

[38] Steffen Smolka, Nate Foster, Justin Hsu, Tobias Kappé, Dexter Kozen, and Alexandra Silva. 2019. Guarded Kleene Algebra with Tests: Verification of Uninterpreted Programs in Nearly Linear Time. In *POPL*. https://doi.org/10.1145/3371129

[39] Steffen Smolka, Praveen Kumar, Nate Foster, Justin Hsu, Dexter Kozen, and Alexandra Silva. 2019. Scalable Verification of Probabilistic Networks. In *PLDI*. https://doi.org/10.1145/3314221.3314639

[40] Steffen Smolka, Praveen Kumar, Nate Foster, Dexter Kozen, and Alexandra Silva. 2017. Cantor Meets Scott: Semantic Foundations for Probabilistic Networks. In *POPL*. https://doi.org/10.1145/3093333.3009843

[41] Alan Tang, Ryan Beckett, Steven Benaloh, Karthick Jayaraman, Tejas Patil, Todd D. Millstein, and George Varghese. 2023. Lightyear: Using Modularity to Scale BGP Control Plane Verification. In *SIGCOMM*. https://doi.org/10.1145/3603269.3604842

[42] Timothy Alberdingk Thijm, Ryan Beckett, Aarti Gupta, and David Walker. 2023. Modular Control Plane Verification via Temporal Invariants. In *PLDI*. https://doi.org/10.1145/3591222

[43] Emina Torlak and Rastislav Bodík. 2013. Growing Solver-aided Languages with Rosette. In *Onward! (SPLASH)*. https://doi.org/10.1145/2509578.2509586

[44] Moshe Y. Vardi and Pierre Wolper. 1986. An Automata-Theoretic Approach to Automatic Program Verification (Preliminary Report). In *LICS*.

[45] Geoffry G. Xie, Jibin Zhan, David A. Maltz, Hui Zhang, Albert Greenberg, Gisli Hjalmtysson, and Jennifer Rexford. 2005. On Static Reachability Analysis of IP Networks. In *INFOCOMM*. https://doi.org/10.1109/INFCOM.2005.1498492

[46] Hongkun Yang and Simon S. Lam. 2016. Real-Time Verification of Network Properties Using Atomic Predicates. *IEEE/ACM Transactions on Networking* 24, 2 (Apr 2016), 887–900. https://doi.org/10.1109/TNET.2015.2398197

[47] Peng Zhang, Xu Liu, Hongkun Yang, Ning Kang, Zhengchang Gu, and Hao Li. 2020. APKeep: Realtime Verification for Real Networks. In *NSDI*.

Received 2023-11-16; accepted 2024-03-31