Active Learning of Symbolic NetKAT Automata[*]

MARK MOELLER, Cornell University, USA
TIAGO FERREIRA, University College London, United Kingdom
THOMAS LU, Cornell University, USA
NATE FOSTER, Cornell University, USA and Jane Street, USA
ALEXANDRA SILVA, Cornell University, USA

NetKAT is a domain-specific programming language and logic that has been successfully used to specify and verify the behavior of packet-switched networks. This paper develops techniques for automatically learning NetKAT models of unknown networks using active learning. Prior work has explored active learning for a wide range of automata (e.g., deterministic, register, Büchi, timed etc.) and also developed applications, such as validating implementations of network protocols. We present algorithms for learning different types of NetKAT automata, including symbolic automata proposed in recent work. We prove the soundness of these algorithms, build a prototype implementation, and evaluate it on a standard benchmark. Our results highlight the applicability of symbolic NetKAT learning for realistic network configurations and topologies.

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

Additional Key Words and Phrases: NetKAT, network verification, automata learning, symbolic automata

ACM Reference Format:
Mark Moeller, Tiago Ferreira, Thomas Lu, Nate Foster, and Alexandra Silva. 2025. Active Learning of Symbolic NetKAT Automata. *Proc. ACM Program. Lang.* 9, PLDI, Article 192 (June 2025), 24 pages. https://doi.org/10.1145/3729295

1 INTRODUCTION

Model-checking is one of the success stories in formal methods, with broad applications in hardware and software. However, model checking requires having accurate models of the systems being verified. These models are usually produced by human experts via a slow and error-prone process.

Active learning is an appealing alternative that can construct an accurate model of a system automatically. Model learning is often used with closed-box systems—i.e., when the source code or inner workings of a system cannot be inspected, but its external behaviors can be observed—and it generates formal models, such as finite automata, that can be used to obtain assurance through testing or verification. In particular, Angluin's $\mathsf{L}^\star$ algorithm, which learns a Deterministic Finite Automaton (DFA) in the so-called Minimally Adequate Teacher (MAT) framework, has inspired nearly 40 years of fruitful research [6, 7, 19, 20, 33].

The $\mathsf{L}^\star$ approach is most applicable when only observations of behavior are available and it is reasonable to assume the system has a finite state model, or can be approximated by such a model. Computer networking is one such domain, which has both of these characteristics. We frequently

[*]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.

Please use nonacm option or ACM Engage class to enable CC licenses
This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2025 Copyright held by the owner/author(s).
ACM 2475-1421/2025/6-ART192
https://doi.org/10.1145/3729295

have only partial information about the policies used to route packets through the network, but we do have the ability to observe packet-level behaviors using command-line tools like `ping` and `traceroute`. At the same time, the programs that execute on network routers are finite state—their behavior is kept intentionally simple to enable them to operate at line rate. Indeed, automata learning has been applied to learning implementations of network protocols [10, 13].

In this paper, we want to go a step further and learn not just end-host protocols, but network-wide behaviors. This would be helpful in many scenarios: inferring a model of a single device where the program and configuration are unavailable, to verify that it conforms to a well-understood model; inferring the topology of the network and check that it conforms to the expected structure; or inferring a model an unknown peer network to verify that it is fulfilling a customer agreement.

Indeed, the networking community has recognized that there is great value in having accurate models of individual devices and the network as a whole. For example, Google employs a small team of verification engineers who develop and maintain models of their data center switches in a domain-specific language [1]. They use these models to find bugs (using fuzzers) and verify properties (using symbolic execution tools). However, maintaining the models turns out to be non-trivial—switch pipelines are moderately complex, and their behavior changes as vendors add new features. Using automata learning, rather than having to expend effort developing these models by hand, they could be generated automatically. Learned models also have the advantage of being *ground-truth* in the sense that they are based on actual observations of system behavior.

This paper presents a framework for automated learning of NetKAT automata. To achieve this goal, we bring together two lines of research: active learning and algebraic approaches to network verification. We develop new automata learning algorithms for the expressive models used in NetKAT, a domain-specific programming language and logic based on Kleene Algebra with Tests [3]. NetKAT allows us to model different aspects of a network at different levels of detail and abstraction, applicable to all the scenarios described above. Moreover, as NetKAT was designed with an automata-theoretic foundation, it is a great fit for active learning. The key challenges, however, are in designing specialized data structures and algorithms that take advantage of NetKAT's algebraic structure and other details of the networking domain, to scale up to real-world networks.

In particular, while NetKAT has an elegant theory based on using packets as actions in the automaton model, for learning network policies, it has an obvious drawback: the space of packets is exponential in the number of header bits—intractably large for all but toy policies. To overcome the large packet space challenge, recent work introduced a symbolic form of NetKAT automata [27]. Crucially, these automata are symbolic in *both* the transition labels and the state space. Hence, these symbolic NetKAT automata are therefore the ideal target for exploring learning.

Symbolic NetKAT automata are reminiscent of Symbolic Finite Automata (SFAs) [8], a fundamental model introduced to deal with infinite or intractably large alphabets in automata. The key idea in SFAs is that rather than transitions being explicitly defined by the singular element of the alphabet they consume, one can have transitions be labeled by predicates that determine the subset of the alphabet that can take a specific transition. SFAs have enjoyed a range of useful applications [30], largely due to their simple theory and efficient decision procedures, and extensions of classic automata learning algorithms have been proposed for subclasses of symbolic automata [5, 9, 11]. Of particular relevance is recent work on learning a large subclass of SFAs [5].

At first glance, prior work on learning SFAs seems to provide the recipe for dealing with NetKAT automata: just compile the NetKAT automata down to DFAs, and use SFA techniques to deal with the large alphabet. Unfortunately, there are several key differences that make this idea a nonstarter: the semantics of NetKAT automata allow transitions to be based on a notion of a *carry-on packet* (or current packet), which means that the naive translation of NetKAT automata to DFAs also triggers

Syntax		**Description**	**Semantics** $[\![p]\!] \subseteq \mathsf{Pk} \cdot (\mathsf{Pk} \cdot \mathsf{dup})^\star \cdot \mathsf{Pk}$
$p, q ::=$	$\bot$	*False*	$\emptyset$
$\mid$	$\top$	*True*	$\{\alpha\alpha \mid \alpha \in \mathsf{Pk}\}$
$\mid$	$f = v$	*Test equals*	$\{\alpha\alpha \mid \alpha.f = v\}$
$\mid$	$f \neq v$	*Test not equals*	$\{\alpha\alpha \mid \alpha.f \neq v\}$
$\mid$	$f \leftarrow v$	*Modification*	$\{\alpha\alpha[f \leftarrow v] \mid \alpha \in \mathsf{Pk}\}$
$\mid$	dup	*Duplication*	$\{\alpha \cdot (\alpha \cdot \mathsf{dup}) \cdot \alpha \mid \alpha \in \mathsf{Pk}\}$
$\mid$	$p + q$	*Union*	$[\![p]\!] \cup [\![q]\!]$
$\mid$	$p \cdot q$	*Sequencing*	$[\![p]\!] \diamond [\![q]\!]$
$\mid$	$p^\star$	*Iteration*	$\bigcup_{n \geq 0} [\![p^n]\!]$ s.t. $p^0 \triangleq \top$, $p^{n+1} \triangleq p \cdot p^n$

Fig. 1. NetKAT syntax and semantics.

a blow-up of *state*. This is precisely the issue addressed in symbolic NetKAT automata [27], and is the reason why we cannot re-use learning algorithms for SFAs with NetKAT.

This paper develops active learning for NetKAT automata, making the following contributions:

- **An expanded theory of NetKAT automata**: Angluin-style learning algorithms crucially rely on the classic Myhill-Nerode theorem, which identifies the states of the minimal automaton with equivalence classes of languages. We present a new Myhill-Nerode theorem for NetKAT (Section 3) that enable us to characterize a *canonical* form for NetKAT automata.
- **A naive learning algorithm**: We give a "first attempt" solution for learning canonical NetKAT automata, and prove its correctness (Section 4). It is naive in that it does not use symbolic techniques, and it therefore has poor performance due to the large "alphabet" and state space.
- **Two algorithms for symbolic NetKAT learning**: To provide scalable learning for NetKAT, we also develop an algorithm for learning symbolic NetKAT automata. We do this in two stages. First, we focus on the subset of NetKAT that does not have the dup primitive (i.e., the dup-free fragment). We give an algorithm for learning symbolic representations of NetKAT programs in this restricted setting (Section 5). Next, using this first symbolic learner as a subroutine for symbolic learning of full-blown NetKAT automata and prove its correctness (Section 6).
- **A prototype OCaml implementation**: We have implemented our symbolic learning algorithms as a prototype to investigate their behavior and properties. We conclude our paper, by describing our implementation (Section 7) and experience learning models for network policies from a standard benchmark suite (Section 7.1).

Omitted proofs can be found in the extended version of this paper [25].

2 BACKGROUND ON NETKAT AND NETKAT AUTOMATA

NetKAT [3] was introduced as a language for reasoning about network routing policies with axiomatic, decidable program equivalence as a design principle. This section briefly reviews NetKAT and NetKAT automata as developed in prior work. We fix a set of packet header fields $F = \{f_1, \dots, f_n\}$, and a finite set of header values V. We write records (finite maps) as a collection of mappings. For example, a *packet* is a finite record assigning values to fields $\alpha = \{f_1 \mapsto v_1, \dots, f_n \mapsto v_n\}$. We reference the field f_1 of packet α by $\alpha.f_1 = v_1$. The set of all packets is denoted $\mathsf{Pk} = F \Mapsto V$.

The syntax and semantics of NetKAT are presented in Figure 1. The basic primitives in NetKAT are packet tests ($f = v$, $f \neq v$) and modifications ($f \leftarrow v$). Program expressions are then compositionally built from tests and packet modifications, using union (+), sequencing (·), and iteration (⋆).

Conditionals and loops can be encoded in the standard way:

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

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 [3]. Note that assignments and tests in NetKAT are always against constant values. This is a key restriction that makes equivalence decidable and aligns the language with the capabilities of network 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 "observe" intermediate packets at internal switches.

The semantics of NetKAT is based on regular sets of *guarded strings* which are elements of the set $\mathsf{GS} = \mathsf{Pk} \cdot (\mathsf{Pk} \cdot \mathsf{dup})^\star \cdot \mathsf{Pk}$. Here, dup can be thought of as delimiting "letters" in the string. Equivalently, we can think of guarded strings as elements of the isomorphic set $\mathsf{Pk} \cdot \mathsf{Pk}^\star \cdot \mathsf{Pk}$ which are strings over the alphabet Pk with at least two letters. Given two guarded strings $\alpha x \beta$ and $\gamma y \xi$ their concatenation, denoted by $\diamond$, is defined only when $\beta = \gamma$:

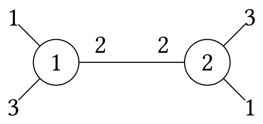
$$\alpha x \beta \diamond \gamma y \xi = \begin{cases} \alpha x y \xi & \beta = \gamma \\ \text{undefined} & \beta \neq \gamma \end{cases} \tag{1}$$

This definition lifts to sets of guarded strings $U, V \subseteq \mathsf{GS}$ pointwise: $U \diamond V = \{u \diamond v \mid u \in U \text{ and } v \in V\}$.

NetKAT's formal semantics, given in Figure 1 associates sets of guarded strings to each expression. The semantics for $\top$ gives all traces containing the input/output packet $\alpha\alpha$, whereas $\bot$ produces no traces. Tests ($f = v$ and $f \neq v$) produce filter traces, depending on whether the test succeeds or fails. Modifications ($f \leftarrow v$) produce a trace with the modified packet. Duplication (dup) produces traces with two copies of the input packet α. 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 concatenation of p, iterated zero or more times.

2.1 Encoding networks in NetKAT

We encode the behavior of networks in NetKAT using a similar approach as in prior work [3, 16, 27, 29]. For example, suppose we have a toy network with two switches with three ports each, connected by a link between both switches' port 2:



We use two special fields sw and pt, which denote a packet's location at a particular switch and port. Using these fields, we can encode the network topology in NetKAT—e.g., the expression,

$$\mathsf{t} \triangleq \mathsf{pt}{=}1 + \mathsf{pt}{=}3 + \mathsf{pt}{=}2 \cdot (\mathsf{sw}{=}1 \cdot \mathsf{sw}{\leftarrow}2 + \mathsf{sw}{=}2 \cdot \mathsf{sw}{\leftarrow}1)$$

Similarly, we can encode routing policies for individual switches in the network above:

$$r \triangleq \mathsf{pt}{=}1 \cdot \mathsf{pt}{\leftarrow}2 + \mathsf{pt}{=}2 \cdot \mathsf{pt}{\leftarrow}1,$$

which encodes the policy: "At either switch, forward packets arriving on port 1 via port 2, and forward packets arriving on port 2 via port 1." This policy drops packets arriving at port 3.

Using these encodings, we can compose the topology (t) and routing policy (r) into one expression capturing the network's *transfer function*: $r \cdot t$. We combine this with ingress and egress predicates p_i and p_f to encode the full end-to-end behavior: $p_i \cdot (r \cdot t \cdot \mathsf{dup})^\star \cdot p_f$. Here we use a dup to denote that we want to log one copy of each packet for each network "hop" (i.e. an update to sw in t). The star indicates that we are interested in all paths allowed by this policy on this topology. The expressions p_i and p_f are predicates which constrain the initial and final packets we are interested in. For example, to investigate all paths by which Switch 1 (Port 1) can reach Switch 2 (Port 1), we might choose $p_i \triangleq \mathsf{sw}{=}1 \cdot \mathsf{pt}{=}1$, and $p_f \triangleq \mathsf{sw}{=}2 \cdot \mathsf{pt}{=}1$. More complicated routing policies may involve other pertinent fields, perhaps dst, encoding a destination switch.

It is the aim of this paper to begin to answer the question: Can we build a model that completely describes this network and its policy by sending packets and observing how they are processed? After building up the component processes, we return to this example in Section 6.8.

2.2 NetKAT Automata

Despite the fact that assignments and tests in NetKAT programs are always against constants, the behavior of NetKAT programs can be complicated, particularly when conditionals, iteration, and multiple packet fields are involved. To provide a more operational view of the semantics, NetKAT automata were introduced by Foster et al. [16] as acceptors of guarded strings.

Definition 2.1. A NetKAT automaton (NKA) is a four-tuple $(S, s_0, \delta, \varepsilon)$, where S is a finite set of states, $s_0 \in S$ is the start state, δ and ε are the transition and observation functions, respectively[1]:

$$\delta\colon S \to \mathsf{Pk} \to \mathsf{Pk} \to S \qquad\qquad \varepsilon\colon S \to \mathsf{Pk} \to \mathsf{Pk} \to 2$$

We will write $\varepsilon_{\alpha\beta}(s)$ as a shorthand for $\varepsilon(s)(\alpha)(\beta)$ and $\delta_{\alpha\beta}(s)$ as a shorthand for $\delta(s)(\alpha)(\beta)$.

Semantically, an NKA $\mathcal{M} = (S, s_0, \delta, \varepsilon)$ accepts a language $\mathcal{L}(\mathcal{M}) \subseteq \mathsf{GS}$ containing all strings $s \in \mathsf{GS}$ for which $\mathcal{M}(s_0, w) = \top$, where:

$$\mathcal{M}(s, \alpha \cdot \beta) = \varepsilon_{\alpha\beta}(s) \qquad\qquad \mathcal{M}(s, \alpha \cdot \beta \cdot \mathsf{dup} \cdot w') = \mathcal{M}(\delta_{\alpha\beta}(q), \beta \cdot w') \tag{2}$$

Note how in the semantics of state s of NetKAT automata we read two packets $\alpha\beta$ moving to state $\delta_{\alpha\beta}(s)$ *but* we retain β in the string that remains! So, unlike traditional regular expressions and automata, NetKAT is stateful—in effect, the packet β has not been fully consumed, but is used as a *carry-on* packet to remember the last step of the automaton's execution. This peculiarity of NetKAT complicates the semantics (i.e., it is not a straightforward language semantics) and designing equivalence and minimization procedures, which have to account for the carry-on packet. We can fold the packet into the state of the automaton and approach the semantics and the problem of learning using classical automata, but this results in enormous state blowup and is therefore impractical. This is the key challenge for designing a learning algorithm we address in this paper.

3 MYHILL-NERODE THEOREM FOR NETKAT AND CANONICAL AUTOMATA

The essential insight of the $\mathsf{L}^\star$ algorithm is that we can identify Myhill-Nerode equivalence classes for DFAs in an incremental fashion. However, to use this idea in the context of NetKAT, we need to develop a NetKAT-specific notion of Myhill-Nerode. This section develops such a notion, which we will use as a basis for a learning algorithm in Section 4. Our approach follows Kozen's more general treatment of Myhill-Nerode for automata on guarded strings [22].

For a language $\mathcal{L}$ over an alphabet Σ, the Myhill-Nerode [28] relation for $\mathcal{L}$ is an equivalence relation on $\Sigma^\star$, written $(\equiv_{\mathcal{L}})$ and defined by:

$$s \equiv_{\mathcal{L}} t \overset{\triangle}{\iff} \forall e \in \Sigma^\star.(se \in \mathcal{L} \iff te \in \mathcal{L})$$

The Myhill-Nerode theorem states that $\mathcal{L}$ is regular iff $\equiv_{\mathcal{L}}$ has finite index. Moreover, the equivalence classes of $\equiv_{\mathcal{L}}$ correspond precisely to the states of the *unique minimal* DFA for $\mathcal{L}$. We would like an analogous characterization for NetKAT, but there are two key challenges: concatenation of guarded strings is not always defined (eq. (1)) and NetKAT's semantics process strings by reading

[1]We use 2 to denote the Booleans, $\{\bot, \top\}$.

two packets at a time–one of which continues as *carry-on* packet for the next step (eq. (2)). These differences result in a subtly different characterization for guarded strings.

We first define the set of guarded string prefixes $\mathsf{Pre} = \mathsf{Pk} \cdot (\mathsf{Pk} \cdot \mathsf{dup})^\star$ and the set of guarded string suffixes $\mathsf{Suf} = (\mathsf{Pk} \cdot \mathsf{dup})^\star \cdot \mathsf{Pk}$. We also define a helper function $\mathsf{last}\colon \mathsf{Pre} \to \mathsf{Pk}$ and a (partial) concatenation operation $\blacklozenge\colon \mathsf{Pre} \times \mathsf{GS} \to \mathsf{GS}$ for prefixes with guarded strings by:

$$\begin{aligned} \mathsf{last}(\alpha) &= \alpha \\ \mathsf{last}(w \cdot \alpha \cdot \mathsf{dup}) &= \alpha \end{aligned} \qquad\qquad p \blacklozenge \alpha \cdot s = \begin{cases} p \cdot s & \mathsf{last}(p) = \alpha \\ \text{undefined} & \text{otherwise} \end{cases}$$

For a language $\mathcal{L} \subseteq \mathsf{GS}$, we define the relation $\equiv_{\mathcal{L}}$ for $s, t \in \mathsf{Pre}$ by:

$$s \equiv_{\mathcal{L}} t \overset{\triangle}{\iff} (\forall e \in \mathsf{GS}.\ s \blacklozenge e \in \mathcal{L} \iff t \blacklozenge e \in \mathcal{L}) \tag{3}$$

The relation $\equiv_{\mathcal{L}}$ is an equivalence on guarded string prefixes.

Lemma 3.1. *The relation $\equiv_{\mathcal{L}}$ in Equation (3) is reflexive, symmetric, and transitive.*

The equivalence classes of the classic Myhill-Nerode relation for $\mathcal{L} \subseteq \Sigma^\star$ correspond exactly to the states of the minimal DFA accepting $\mathcal{L}$. However, this result does not immediately transfer to NetKAT. We need to define a simpler form of automata that hides the carry-on packet in the state—this will enable us to precisely capture the fact that the equivalence classes of $\equiv_{\mathcal{L}}$ can only relate prefixes s and t when $\mathsf{last}(s) = \mathsf{last}(t)$.[2]

Definition 3.2. A *Packet-state* NetKAT automaton (PNKA) is a four-tuple $(Q, q_0, \partial, \lambda)$, where Q is a finite set of states, $q_0 : \mathsf{Pk} \to Q$ is the start state, $\partial\colon Q \to \mathsf{Pk} \to Q$ is the transition function, and $\lambda\colon Q \to \mathsf{Pk} \to 2$ is the observation function.

The functions ∂ and λ must satisfy the *spelling* property: for any states $q, q' \in Q$ and packets $\alpha, \beta \in \mathsf{Pk}$, if $\partial_\alpha(q) = \partial_\beta(q')$, $q_0(\alpha) = q_0(\beta)$, or $q_0(\alpha) = \partial_\beta(q)$, then $\alpha = \beta$.

The spelling property ensures we cannot arrive at the same state via two different packets. Formally, there is a function $\mathsf{spell}\colon Q \to \mathsf{Pk}$ that outputs the packet of all transitions into each state.

A PNKA $\mathcal{M} = (Q, q_0, \partial, \lambda)$ accepts a language $\mathcal{L}(\mathcal{M}) \subseteq \mathsf{GS}$ consisting of all strings $\alpha \cdot w \in \mathsf{GS}$ satisfying $\mathcal{M}(q_0(\alpha), w) = \top$, where:

$$\mathcal{M}(q, \beta) = \lambda_\beta(q) \qquad \mathcal{M}(q, \beta \cdot \mathsf{dup} \cdot w') = \mathcal{M}(\partial_\beta(q), w')$$

As the name suggests, the distinguishing feature of a PNKA is that the transition function and observation function operate on a single packet: the other packet is "hidden in the state." Although PNKA satisfy additional restrictions compared to standard NetKAT automata, it is important to note that they recognize exactly the same sets of guarded strings.

Theorem 3.3. *Let $\mathcal{L} \subseteq \mathsf{GS}$. Then $\mathcal{L}$ is accepted by a NKA if and only if it is accepted by a PNKA.*

Now that we have established that PNKA are a suitable representation of NKA we are ready to state the Myhill-Nerode theorem for NetKAT. This theorem ensures PNKA have properties that are desirable for automata learning.

Definition 3.4. Given a language $\mathcal{L}$ we define $\mathcal{P}_{\equiv_{\mathcal{L}}} = (Q, q_0, \partial, \lambda)$ by:

$$Q = \{[s]_{\equiv_{\mathcal{L}}} \mid s \in \mathsf{Pre}\} \qquad q_0(\alpha) = [\alpha]_{\equiv_{\mathcal{L}}} \qquad \partial_\alpha([s]_{\equiv_{\mathcal{L}}}) = [s \cdot \alpha \cdot \mathsf{dup}]_{\equiv_{\mathcal{L}}} \qquad \lambda_\alpha([s]_{\equiv_{\mathcal{L}}}) = s \cdot \alpha \in \mathcal{L}$$

Theorem 3.5 (Myhill-Nerode for NetKAT). *For a given language $\mathcal{L} \subseteq \mathsf{GS}$:*

(1) $\mathcal{L}$ is regular if and only if $\equiv_{\mathcal{L}}$ has finite index.
(2) For regular $\mathcal{L}$, a minimal PNKA accepting $\mathcal{L}$ has $|\equiv_{\mathcal{L}}|$ states.
(3) Any PNKA accepting $\mathcal{L}$ with $|\equiv_{\mathcal{L}}|$ states is isomorphic to $\mathcal{P}_{\equiv_{\mathcal{L}}}$.

[2]Note that we interpret ($\iff$) to hold when both sides are undefined or both defined and equal, but it does *not* hold when only one side is undefined (regardless of the other side).

4 LEARNING CANONICAL NETKAT AUTOMATA

In this section, we develop a learning algorithm for *canonical* NetKAT automata (i.e., minimal PNKA). This algorithm is primarily of theoretical interest, as canonical automata will have a large number of states in general. However, the core ideas of this approach will guide us in designing a more practical algorithm in Section 6 for learning *symbolic* NetKAT automata.

Our algorithm resembles $\mathsf{L}^\star$, but we cannot use the same state for different "carry-on" packets in a PNKA. Accommodating this restriction requires some additional structure.

4.1 The PNKA Teacher

Just as in Angluin's MAT, the MAT framework for NetKAT automata assumes a teacher that can answer two types of queries: membership queries (i.e., whether a given guarded string is part of the target language or not) and equivalence queries (i.e., whether a given hypothesis automaton describes the target language, and returning a counter-example if not).

Definition 4.1 (MAT Interface). The MAT interface for Packet-state NetKAT automata is as follows:

$$\mathsf{mem}_{\mathsf{PNKA}} : \mathsf{GS} \to 2 \qquad \mathsf{equiv}_{\mathsf{PNKA}} : \mathsf{PNKA} \to 1 + \mathsf{GS}$$

The teacher knows the target language, $\mathcal{L} \subseteq \mathsf{GS}$. We have $\mathsf{mem}_{\mathsf{PNKA}}(w) = \top$ iff $w \in \mathcal{L}$, and $\mathsf{equiv}_{\mathsf{PNKA}}(\mathcal{H}) = \top$ if $\mathcal{L}(\mathcal{H}) = \mathcal{L}$. Otherwise, $\mathsf{equiv}_{\mathsf{PNKA}}(\mathcal{H}) = w \in \mathcal{L} \oplus \mathcal{L}(\mathcal{H})$. In the rest of this section we develop a naive algorithm for learning a canonical PNKA from this type of teacher.

4.2 The Observation Table

The central abstraction used in our learning algorithm is an *observation table*. This data structure keeps track of the knowledge the learner has acquired, and it provides the basis for constructing hypothesis automata. The shape of the table is similar to the classic observation table used in $\mathsf{L}^\star$, but we need to adjust it to the specific nature of NetKAT automata. In particular, we organize the observation table as a collection of smaller tables, one for each packet in Pk.

Definition 4.2 (Packet Table). For a packet $\alpha \in \mathsf{Pk}$, a packet table is a triple $(S_\alpha, E_\alpha, T_\alpha)$, where:

- $S_\alpha \subseteq \{p \mid \mathsf{last}(p) = \alpha, p \in \mathsf{Pre}\}$ is a set of *access prefixes.*
- $E_\alpha \subseteq \mathsf{Suf}$ is a set of *distinguishing suffixes.*
- $T_\alpha \colon (S_\alpha \cup S'_\alpha) \times E_\alpha \to 2$ is such that $T_\alpha(s, e) = \top$ if $s \cdot e \in \mathcal{L}$, and $T_\alpha(s, e) = \bot$ otherwise.

We maintain $S'_\alpha \triangleq \bigcup_{\beta \in \mathsf{Pk}} S_\beta \cdot (\alpha \cdot \mathsf{dup})$ as the set of $(\alpha \cdot \mathsf{dup})$ *extensions* of all packet table sets S_β.

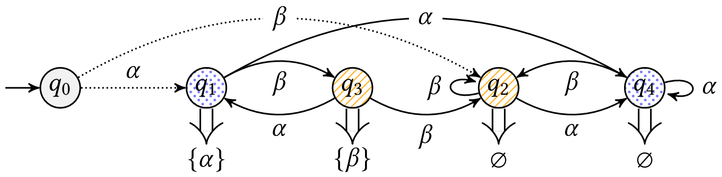
Definition 4.3 (Observation Table). An observation table T is an α-indexed set of Packet Tables, $(S_\alpha, E_\alpha, T_\alpha)$, consisting of one packet table per packet in Pk.

Remark 1. *In $\mathsf{L}^\star$, the extension part of the table is usually written $S \cdot \Sigma$, and computes the one letter extension of each string representing a state—i.e., the transitions of an automaton. We require a similar functionality in our case, albeit adjusted to the specific details of packet tables. In particular, for a packet table $(S_\alpha, E_\alpha, T_\alpha)$, the set S'_α is maintained to have the $(\alpha \cdot \mathsf{dup})$ extension of every state in every packet table. This allows T_α to hold information on every α transition between any pair of states.*

Access Prefixes. Access prefixes describe states of the target automaton—the state corresponding to a given access prefix is the one the automaton would land on after processing that prefix. Prefixes comprising a single packet naturally represent the start state, while prefixes comprising two packets (and one dup) represent the states reached by reading a given packet from the start state, and so on.

Distinguishing Sequences. Distinguishing sequences, on the other hand, elicit specific behavior from an arbitrary state. Note that because the prefixes come from Pre and distinguishing sequences from Suf, concatenation is always defined and forms a guarded string. Distinguishing sequences can also be understood as the suffixes e in the Myhill-Nerode relation of Section 3. In $\mathsf{L}^\star$, the set of distinguishing suffixes is maintained to be suffix-closed by the algorithm. A similar property holds here, but only for the observation table as a whole, not each individual packet table.

Example 4.4. Consider the PNKA $\mathcal{M}$ below over the packet space $\mathsf{Pk} = \{\{f \mapsto 1\}, \{f \mapsto 2\}\}$, which we abbreviate to α and β, respectively. States from equivalence classes ending in α are colored orange and lined, while states from equivalence classes ending in β are colored blue and dotted. The dotted transitions from the start state denote its special type $q_0 \colon \mathsf{Pk} \to Q$ in the automaton.



Note that there are infinitely many possible access prefixes and distinguishing sequences, due to the cyclic structure of the automaton. Some access prefixes for q_1 are α and $\alpha \cdot \beta \cdot \mathsf{dup} \cdot \alpha$. An access prefix for q_3 is $\alpha \cdot \beta \cdot \mathsf{dup}$. Finally, β is a distinguishing sequence for q_1 and q_3, as $\mathcal{M}(q_0(\alpha), \beta) = \bot$ but $\mathcal{M}(q_0(\alpha), \beta \cdot \mathsf{dup} \cdot \beta) = \top$.

Closed and Consistent Observation Tables. In $\mathsf{L}^\star$, the main loop of the algorithm makes membership queries to construct a closed table, from which a hypothesis can be made. We define an analogous concept here, also in terms of the row function, just as in $\mathsf{L}^\star$. Namely, we define a primitive row function with type $\mathsf{row}_\alpha : S_\alpha \cup S'_\alpha \to E_\alpha \to 2$, defined by $\mathsf{row}_\alpha(s)(e) = T_\alpha(s, e)$. Clearly all we have done here is to curry T_α—but the currying is essential! We will partially apply row_α to a given string s and consider the resulting functions $E_\alpha \to 2$. In this way, we distinguish states by viewing their behavior in terms of the suffixes we have seen for each packet.

Definition 4.5 (Closed Table). An observation table T is *closed*, if for each $\alpha \in \mathsf{Pk}$ and for each $s' \in S'_\alpha$, then $\mathsf{row}_\alpha(s') = \mathsf{row}_\alpha(s)$ for some $s \in S_\alpha$.

Definition 4.6 (Consistent Table). An observation table T is *consistent*, if for each $\alpha, \beta \in \mathsf{Pk}$ and for each $s, s' \in S_\alpha$, then $\mathsf{row}_\alpha(s) = \mathsf{row}_\alpha(s') \Rightarrow \mathsf{row}_\beta(s \cdot \beta \cdot \mathsf{dup}) = \mathsf{row}_\beta(s' \cdot \beta \cdot \mathsf{dup})$.

The closedness property can be checked locally to each packet table (provided that all the required extensions to each prefix by each packet have all already been entered into T), while the consistency property must be checked globally, since successors to rows in one table may be in another table.

We use the observation table in the same way as in $\mathsf{L}^\star$: by making membership queries until the table is closed and consistent, ensuring that the construction of the next PNKA to conjecture is well-defined. In Figure 3, we show how to make the table closed and consistent. The function extend performs membership queries as needed so that every T_α is defined for its required domain.

4.3 The PNKA Learning Algorithm

Our NKA learning algorithm works by iteratively expanding an observation table, which is then the basis from which a hypothesis is constructed. If the hypothesis is correct then the learner terminates. Otherwise, a counterexample from the equivalence oracle is used to refine the tables. The main routine of the algorithm (Figure 2) follows the classic learning loop of MAT-based learners.

```
$T \leftarrow \{\alpha \mapsto (S_\alpha = \{\alpha\}, E_\alpha = \mathsf{Pk}, T_\alpha = \{\}) \mid \alpha \in \mathsf{Pk}\}$
**while** $\top$ **do**
    **while** $T$ not closed and consistent **do**
        $T \leftarrow \mathsf{mkConsistent}(\mathsf{mkClosed}(T))$
    $\mathcal{H} \leftarrow \mathsf{hyp}_{\mathsf{PNKA}}(T)$
    $c \leftarrow \mathsf{equiv}_{\mathsf{PNKA}}(\mathcal{H})$
    **if** $c = \top$ **then**
        **return** $\mathcal{H}$
    **else**
        $T \leftarrow \mathsf{update}(T, c)$
```

```
**Input:** Table $T$, counterex. $c \in \mathsf{GS}$
**Output:** $T$ updated in place with $c$.

**for** $s \in \mathsf{Pre}$, $e \in \mathsf{Suf}$ such that $c = s \cdot e$ **do**
    $S_{\mathsf{last}(s)} \leftarrow S_{\mathsf{last}(s)} \cup \{s\}$
**return** $\mathsf{extend}(T)$
```

Fig. 2. Algorithm for learning canonical automata (left) and subroutine for update (right).

```
**if** $T_\alpha$ is not closed **then**
    let $s' \in S'_\alpha$ such that
      $\mathsf{row}_\alpha(s') \notin \{\mathsf{row}_\alpha(s) \mid s \in S_\alpha\}$.
    $S_\alpha \leftarrow S_\alpha \cup \{s'\}$
    **return** $\mathsf{extend}(T)$
```

```
**if** $T$ is not consistent **then**
    let $e \in E_\beta$ such that for some $s, s' \in S_\alpha$, $\mathsf{row}_\alpha(s) = \mathsf{row}_\alpha(s')$,
      but $T_\beta(s \cdot \beta \cdot \mathsf{dup}, e) \neq T_\beta(s' \cdot \beta \cdot \mathsf{dup}, e)$.
    $E_\alpha \leftarrow E_\alpha \cup \{\beta \cdot \mathsf{dup} \cdot e\}$
    **return** $\mathsf{extend}(T)$
```

Fig. 3. Subroutines used in PNKA learning (fig. 2): mkClosed : $T \to T$ (left) and mkConsistent : $T \to T$ (right).

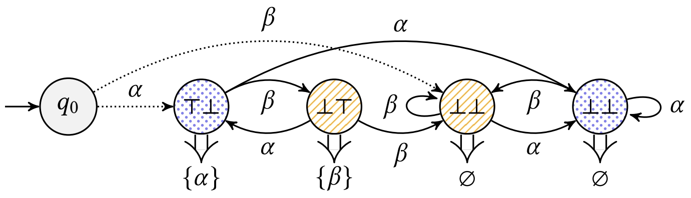
For the update operation, we first break the counterexample string at each possible location. Each break results in a prefix ending in some packet α, which we add to S_α.

Finally, constructing the hypothesis once the table is closed and consistent is again similar to $\mathsf{L}^\star$, except that we still need to keep the state space associated with each table separate. To make this clear in the formalism, states themselves are a pair of type $\mathsf{Pk} \times (E_\alpha \to 2)$: a packet α, and a row vector of T_α. Given an observation table T, we construct $\mathsf{hyp}_{\mathsf{PNKA}}(T) \triangleq (Q, q_0, \delta, \varepsilon)$, where:

$$Q = \biguplus \{(\alpha, \mathsf{row}_\alpha(s)) \mid s \in S_\alpha, \alpha \in \mathsf{Pk}\} \qquad \delta(\alpha, \mathsf{row}_\alpha(s))(\beta) = (\beta, \mathsf{row}_\beta(s \cdot \beta \cdot \mathsf{dup}))$$
$$q_0(\alpha) = (\alpha, \mathsf{row}_\alpha(\alpha)) \qquad \varepsilon(\alpha, \mathsf{row}_\alpha(s))(\beta) = T_\alpha(s, \beta)$$

Example 4.7. Were we to select the PNKA of Example 4.4 as the target, we would have the following final packet tables and hypothesis:

α	α	β
α	$\top$	$\bot$
$\alpha\alpha$	$\bot$	$\bot$
$\alpha\alpha$	$\bot$	$\bot$
$\alpha\alpha\alpha$	$\bot$	$\bot$
$\beta\alpha$	$\bot$	$\bot$
$\alpha\beta\alpha$	$\top$	$\bot$



β	α	β
β	$\bot$	$\bot$
$\alpha\beta$	$\bot$	$\top$
$\alpha\beta$	$\bot$	$\top$
$\alpha\alpha\beta$	$\bot$	$\bot$
$\beta\beta$	$\bot$	$\bot$
$\alpha\beta\beta$	$\bot$	$\bot$

4.4 Correctness of the Canonical Learner

Next, we prove that the canonical automaton learner is correct. We start by showing that the learner produces valid conjectures. For succinctness, we will assume the observation tables used to build hypotheses are closed and consistent, as this follows from the definition of the learning algorithm.

Lemma 4.8. *The hypothesis automaton,* $\mathsf{hyp}_{\mathsf{PNKA}}(T)$*, is well defined for any Observation Table* T.

The next lemma says that the learner only conjectures minimal (i.e., canonical) automata.

Lemma 4.9. *For any hypothesis PNKA $\mathcal{H} = (Q, q_0, \partial, \lambda)$ produced by the learner, and for any two prefixes $s, t \in \mathsf{Pre}$, we have that $s \equiv_{\mathcal{H}} t$ iff $s \equiv_{\mathcal{L}(\mathcal{H})} t$.*

The next lemma says that the Myhill-Nerode relation for the target language always refines the access relation for the learner's conjecture. In other words, the learner can only be wrong by failing to distinguish two states that need to be different.

Lemma 4.10. *Suppose the teacher holds $\mathcal{L}$. Then for each hypothesis PNKA $\mathcal{H}$ of the learner, we have that $(\equiv_{\mathcal{L}})$ refines $(\equiv_{\mathcal{L}(\mathcal{H})})$ in the sense that for prefixes $s, t \in \mathsf{Pre}$, then $s \equiv_{\mathcal{L}} t$ implies $s \equiv_{\mathcal{L}(\mathcal{H})} t$.*

Additionally, hypotheses remain faithful to the data in the observation table they originate from.

Theorem 4.11. *Given a closed, consistent table T, $\mathcal{H} \triangleq \mathsf{hyp}_{\mathsf{PNKA}}(T)$ agrees on every cell of the packet tables. That is, $T_\gamma(\alpha \cdot s, e) = \mathcal{H}(q_0(\alpha), s \cdot e)$ for any $\alpha \cdot s \in S_\gamma \cup S'_\gamma$, $e \in E_\gamma$, and $\gamma \in \mathsf{Pk}$.*

Finally, each counterexample results in a conjecture $\mathcal{H}$, which has at least one more state than the previous conjecture:

Lemma 4.12. *Fix the target language $\mathcal{L}$, and suppose a learner conjectures a PNKA $\mathcal{H}$, and we get a counterexample $c = \mathsf{equiv}_{\mathsf{PNKA}}(\mathcal{H})$, i.e. $c \in \mathcal{L} \oplus \mathcal{L}(\mathcal{H})$. Then the next conjecture by the learner, $\mathcal{H}'$, will have at least one additional state compared to $\mathcal{H}$.*

Theorem 4.13. *The algorithm in Figure 2 terminates with the correct automaton.*

Proof. It follows that if the learner terminates then the automaton is correct, as we only terminate upon a successful equivalence query. Because the Myhill-Nerode relation $(\equiv_{\mathcal{L}})$ for the language $\mathcal{L}$ refines those of the hypotheses (Lemma 4.10), then once we conjecture a hypothesis H for which $|\equiv_{\mathcal{L}}| = |\equiv_{\mathcal{H}}|$, we know the automaton must be correct by Theorem 3.5. By Lemma 4.12, we make progress toward this bound with every counterexample, completing the proof. □

5 LEARNING SYMBOLIC PACKET PROGRAMS (SPPS)

Before we present our algorithm for learning symbolic NetKAT automata, we first develop the core techniques needed to handle a restricted case: dup-free NetKAT programs. We then use the learner we develop in this restricted setting as the central component of a learner that handles full NetKAT. In addition to its role in the full automaton learner, however, this algorithm can be used directly whenever input-output packet pairs are enough to capture the relevant behavior of a system, such as a single closed-box device, or the single-step transfer function of a network.

5.1 Background

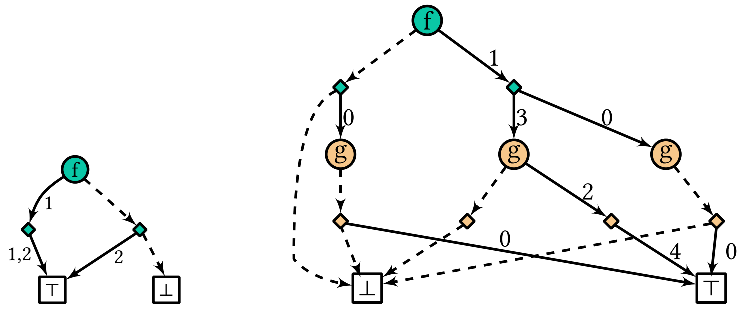
Moeller et al. [27] introduced a set of techniques for reasoning about NetKAT *symbolically*. Their representation first develops a data structure based on Binary Decision Diagrams (BDDs) for the dup-free fragment of NetKAT which they call a Symbolic Packet Program (SPP).

$$p \in \mathsf{SPP} ::= \bot \mid \top \mid \mathsf{SPP}(f, \{\ldots, v_i \mapsto \{\ldots, w_{ij} \mapsto q_{ij}, \ldots\}, \ldots\}, \{\ldots, w_i \mapsto q_i, \ldots\}, q)$$

SPPs have two base cases, $\top$ and $\bot$, which represent the corresponding NetKAT expressions. The third case of an SPP represents testing a field f of an input packet against a series of values $v_0, \ldots, v_i, \ldots, v_n$, with a default case $f \neq v_0 \cdots f \neq v_n$. However, instead of continuing recursively after the test, SPPs nondeterministically 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. Semantically this third case of an SPP has the semantics of the NetKAT expression:

$\sum_i(f = v_i \cdot \sum_j(f \leftarrow w_{ij} \cdot q_{ij})) \;+\; (\prod_i f \neq v_i) \cdot (\sum_i f \leftarrow w_i \;+\; (\prod_i f \neq w_i) \cdot q)$, where f is a field, v_i, w_{ij}, w_i are values, and q_{ij}, q_j, q are SPPs.

Example 5.1. On the left, we draw the SPP corresponding to the NetKAT expression $f = 1 + f \leftarrow 2$, and on the right the one for $(f \leftarrow 0 \cdot g \leftarrow 0) + (f = 1 \cdot g = 2 \cdot f \leftarrow 3 \cdot g \leftarrow 4)$:



In this paper, we write the semantics of SPPs as a function $[\![\cdot]\!] : \mathsf{SPP} \to \mathsf{Pk} \times \mathsf{Pk} \to \mathbb{2}$. This is consistent with the semantics given above as the fragment of $\mathsf{Pk} \cdot (\mathsf{Pk} \cdot \mathsf{dup})^\star \cdot \mathsf{Pk}$ that is reachable without dup is precisely $\mathsf{Pk} \cdot \mathsf{Pk}$. We represent SPPs as directed acyclic graphs, similarly to BDDs. Vertices are labeled with the test field. Solid arrows are labeled with a number encoding the test value, and dashed arrows represent a default case. The sinks of the graph are labeled with $\top$ or $\bot$.

By removing extraneous branches and standardizing between some equivalent forms, SPPs can be made canonical in the sense that for any subset of $\mathsf{Pk} \cdot \mathsf{Pk}$ there is a unique normal-form SPP. In addition, SPPs are closed under all of the NetKAT operations (in Figure 1), as well as difference, symmetric difference, and intersection.

5.2 The SPP Teacher

We first develop a MAT-based framework (as in Section 4) for learning SPPs. The SPP teacher holds a set of guarded strings represented by a dup-free NetKAT expression (i.e., guarded strings of length two, or, pairs of packets). As before, the teacher answers membership and equivalence queries:

$$\mathsf{mem}_{\mathsf{SPP}} : \mathsf{Pk} \times \mathsf{Pk} \to 2 \qquad \mathsf{equiv}_{\mathsf{SPP}} : \mathsf{SPP} \to 1 + \mathsf{Pk} \times \mathsf{Pk}$$

For a membership query, given a pair of packets, the teacher returns $\top$ if the pair is in its language, and $\bot$ otherwise. For an equivalence query, given a SPP h the teacher returns $\top$ if h has the semantics of the target language, otherwise, the teacher returns a pair of packets in the symmetric difference between $[\![h]\!]$ and the target, i.e., a counterexample to the correctness of h.

If the teacher has an SPP for the target program, the implementation of both queries is straightforward: for a membership query we check that the given packet pair is in the semantics of the target (i.e., is a path from root to $\top$), and for the equivalence query, we take the symmetric difference between the target and the hypothesis. If the resulting SPP is $\bot$ we are done; otherwise we choose a counterexample packet pair by choosing a path to $\top$ in the resulting SPP.

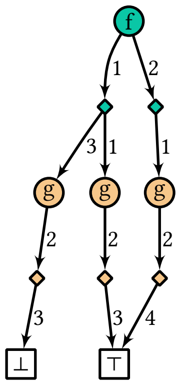
5.3 Evidence Packet Programs

The learner we present below proceeds in a similar fashion to $\mathsf{L}^\star$ [4] or its variants (e.g., KV [20]). The learner maintains a data structure with all of the evidence received from the teacher, and it build successive queries and conjectures guided by this data structure. For learning SPPs, the data structure is called an Evidence Packet Program (EPP):

$$e \in \mathsf{EPP} ::= \top \mid \bot \mid F \times (V \mapsto (V \mapsto \mathsf{EPP}))$$

As with SPPs, we require the fields to appear in a fixed order as we descend the tree structure of an EPP. Unlike SPPs, we will maintain that every field is present (i.e. the $\top$ and $\bot$ leaves all have the same depth). The other distinction from SPPs is that every edge label in an EPP has a concrete value; i.e., there are no default cases. The EPP is essentially a trie containing example packet pairs, where the values are paired by field, as in SPPs.

An example EPP containing the packet pair $(\{f \mapsto 1, g \mapsto 2\}, \{f \mapsto 3, g \mapsto 3\})$, labeled $\bot$, and the pairs $(\{f \mapsto 1, g \mapsto 2\}, \{f \mapsto 1, g \mapsto 3\})$ and $(\{f \mapsto 2, g \mapsto 2\}, \{f \mapsto 1, g \mapsto 4\})$, labeled $\top$, is shown on the right.



We define the semantics of an EPP as $[\![\cdot]\!] \colon \mathsf{EPP} \to \mathsf{Pk} \times \mathsf{Pk} \rightharpoonup 2$, which, given a packet pair, gives the label of the pair in the EPP.

$$[\![\top]\!](\alpha,\beta) \triangleq \top \qquad [\![\bot]\!](\alpha,\beta) \triangleq \bot$$

$$[\![(f,e)]\!](\alpha,\beta) \triangleq \begin{cases} [\![e[\alpha.f][\beta.f]]\!](\alpha,\beta) & \text{if } \alpha.f \in \mathsf{keys}(e) \wedge \beta.f \in \mathsf{keys}(e[\alpha.f]) \\ \text{undefined} & \text{otherwise} \end{cases}$$

The only pairs that we add to an EPP come from membership queries. Hence, we maintain by construction the invariant that $[\![e]\!]$ is always a restriction of the semantics of the target SPP, $[\![s]\!]$, in the sense that whenever $[\![e]\!](\alpha,\beta) = \ell$, it is also the case that $[\![e]\!](\alpha,\beta) = \ell$.

5.4 The SPP Learning Algorithm

The learner executes the following steps in a loop: (i) convert the EPP to an SPP (ii) conjecture an SPP, and (iii) update the EPP with the counterexample received from the teacher, or terminate if the conjecture was correct. The pseudocode for the learner appears in Figure 4. In the rest of this section, we further detail the learner's subroutines.

```
$e \leftarrow \bot$
$c \leftarrow \mathsf{equiv}_{\mathsf{SPP}}(\bot)$
while $c \neq \top$ do
    $e \leftarrow \mathsf{update}(e)(c)(\mathsf{mem}_{\mathsf{SPP}}(c))$
    $h \leftarrow \mathsf{hyp}_{\mathsf{SPP}}(e)$
    $c \leftarrow \mathsf{equiv}_{\mathsf{SPP}}(h)$
return $h$
```

Fig. 4. Algorithm for learning SPPs.

The central function of the learning process the conversion function from EPPs to a hypothesis SPPs: $\mathsf{hyp}_{\mathsf{SPP}} \colon \mathsf{EPP} \to \mathsf{SPP}$, see Figure 5. It works in bottom-up fashion, following the structure of the EPP. At each (f_i, e), it selects one of the bindings of e to represent the default case in the SPP. It uses the same value for the default assignment case. Ideally, the selected binding will correspond to a value that classifies the largest fraction of the evidence in the EPP—a default case with this property helps keep the conjectured SPP small. The mechanism for making this choice is to choose the branch that results in the smallest SPP after canonicalization. In the pseudocode below, the function $\mathsf{select}(v)$ builds the SPP that would result from v being chosen as the default case, and $\mathsf{hyp}_{\mathsf{SPP}}$ works by taking the minimum over $\mathsf{select}(v)$ for all keys v for each level of the EPP. Note this process does not, in general, result in the *smallest possible* SPP consistent with the EPP—it turns out it is NP-hard to find smallest SPPs in general (see Section 5.5). To quantify the size of an SPP, we use the following function $\mu \colon \mathsf{SPP} \to \mathbb{N}$:

$$\mu(\top) = \mu(\bot) = 0 \qquad \mu(SPP(f,b,m,d)) = \#\text{keys, recursively, in } b, m, d$$

That is, this cost function is simply a sum of the keys in the maps at each level of the SPP, which is a reasonable approximation of the memory footprint of an SPP in a memoized implementation. When we take the minimum of SPPs in the definition below, it is with respect to μ. Note that spp is a "smart constructor" that performs canonicalization of the SPPs. We rely on canonicalization to remove redundant test branches that have the same behavior as the selected default case.

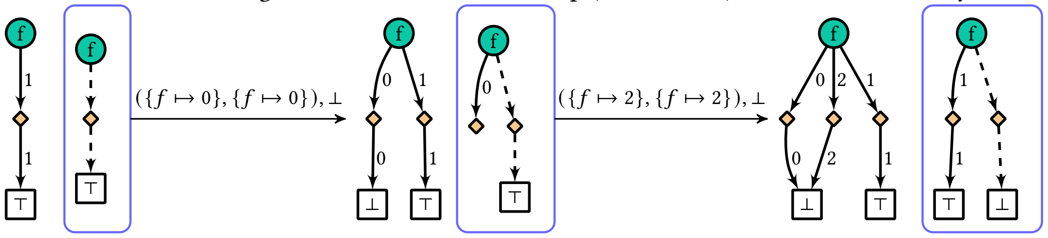
$$
\begin{aligned}
&\mathsf{hyp}_{\mathsf{SPP}}\ \bot \triangleq \bot \qquad \mathsf{hyp}_{\mathsf{SPP}}\ \top \triangleq \top \\
&\mathsf{hyp}_{\mathsf{SPP}}(\mathsf{EPP}(f, e)) \triangleq \min_{v \in \mathsf{keys}(e)} \mathsf{select}(v), \text{where:} \\
&\quad \mathsf{select}(v) \triangleq \mathsf{spp}(f, b, m, d), \text{where:} \\
&\qquad b = \{v' \mapsto \{v'' \mapsto \mathsf{hyp}_{\mathsf{SPP}}\ e' \mid v'' \mapsto e' \in m'\} \mid v' \mapsto m' \in e, v' \neq v\} \\
&\qquad m = \{v' \mapsto \mathsf{hyp}_{\mathsf{SPP}}\ e' \mid v' \mapsto e' \in e[v], v' \neq v\} \\
&\qquad d = \begin{cases} \mathsf{hyp}_{\mathsf{SPP}}\ e[v][v] & \text{if } v \in \mathsf{keys}(e) \text{ and } v \in \mathsf{keys}(e[v]) \\ \bot & \text{otherwise} \end{cases}
\end{aligned}
$$

Fig. 5. Definition of $\mathsf{hyp}_{\mathsf{SPP}}$, for generalizing EPPs to SPPs.

The $\mathsf{update} \colon \mathsf{EPP} \to \mathsf{Pk} \times \mathsf{Pk} \to 2 \to \mathsf{EPP}$ operation used in in Figure 4 just adds an example pair to the trie. We assume an implementation that, whenever $e' = \mathsf{update}\ e(\alpha, \beta)\ \ell$, then the label for (α, β) is updated in e' and no other pairs are affected:

$$[\![e']\!](\alpha, \beta) = \ell \wedge \forall(\alpha', \beta') \neq (\alpha, \beta) : [\![e]\!](\alpha', \beta') = \ell' \Rightarrow [\![e']\!](\alpha', \beta') = \ell'.$$

Example 5.2. Consider applying the algorithm in Figure 4 to the NetKAT expression $f = 1$. The SPPs wrapped in blue boxes below indicate the conjectures h made in each iteration, while the EPPs drawn to the left of the SPPs indicate the set of evidence e according to which the conjectures are made. The labels on the arrows indicate the counterexamples given by the teacher in each iteration. The initial state of the algorithm before the while loop (when $e = \bot$) is omitted for brevity.



The SPP learner in Figure 4 eventually learns the correct SPP held by the teacher. The following lemma shows that SPPs constructed from EPPs are consistent with all the packet pairs in the EPP:

Lemma 5.3. *Let e be an* EPP, *with the number of fields $|F| \geq 1$. For any $v \in V$, and let $s = \mathsf{SPP}(f, b, m, d) = \mathsf{select}\ v$ for e. Then for all (α, β) such that $[\![e]\!](\alpha, \beta) = \top$, we have $[\![s]\!](\alpha, \beta) = \top$, and for all (α, β) such that $[\![e]\!](\alpha, \beta) = \bot$, we have $[\![s]\!](\alpha, \beta) = \bot$.*

In addition, the learning algorithm terminates ensuring soundness.

Theorem 5.4. *The algorithm in Figure 4 terminates with a correct SPP.*

5.5 Finding small SPPs is NP-hard

Typically when developing learning algorithms, a learner usually tries to make conjectures as small as possible, as smaller conjectures usually generalize better and therefore result in more informative counterexamples. However, our $\mathsf{hyp}_{\mathsf{SPP}}$ does not produce SPPs which are minimal in general. It turns out that finding small SPP consistent with a given EPP is NP-hard:

Definition 5.5. The Small Consistent SPP Problem, SmallSPP, is to determine, given an EPP e and $k \geq 0$, whether there exists an SPP s consistent with e such that $\mu(s) \leq k$.

Theorem 5.6. *SmallSPP is NP-hard.*

Proof sketch. By reduction from IndependentSet, which is to determine given an undirected graph $G = (V, E)$ and $k \geq 0$, whether there is a set $V' \subseteq V$ such that $|V'| \geq k$ and for all $v_1, v_2 \in V'$, $(v_1, v_2) \notin E$. Given a graph (V, E), we build an EPP with fields f_1, f_2 and values $V \cup V \times V \cup \{1, 2\}$ as follows. First define two small EPPs:

$$e_\top = \mathsf{EPP}(f_2, \{1 \mapsto \{2 \mapsto \top\}, 2 \mapsto \{1 \mapsto \bot\}\}) \qquad e_\bot = \mathsf{EPP}(f_2, \{1 \mapsto \{2 \mapsto \bot\}, 2 \mapsto \{1 \mapsto \top\}\}),$$

Then we define our EPP by:

$$\mathsf{to\_epp}(G) = \mathsf{EPP}(f_1, \{v_1 \mapsto \{\{v_1, v_2\} \mapsto \mathsf{h}(v_1, v_2) \mid v_2 \in V, v_1 \neq v_2\} \mid v_1 \in V\})$$

where $h(v_1, v_2)$ satisfies that, for any $v_1, v_2 \in V$: $h(v_1, v_2) = h(v_2, v_1) = e_\top$ if $\{v_1, v_2\} \notin E$, and $h(v_1, v_2) = e_\top$ and $h(v_2, v_1) = e_\bot$ otherwise. It can be shown that every SPP consistent with this EPP corresponds to an independent set of G (the independent set is the vertices not chosen for the test branches of the SPP). As a result, the smallest SPP corresponds to the largest independent set. □

6 LEARNING SYMBOLIC NETKAT AUTOMATA

The learner in Section 4 is appealing for its simplicity, but it has a critical limitation that restricts its applicability in practice. If we require states to be separated by packet, then PNKAs are necessarily much larger than necessary. In addition, the learning algorithm itself must repeatedly iterate through the entire space of packets to update the observation table. To improve on this state of affairs, we now present an algorithm, $\mathsf{NKL}^\star$, for learning *symbolic* NetKAT automata, which uses the SPP learner of Section 5 as a subroutine. The advantage to symbolic automata is that they are more compact, as they capture behaviors across sets of packets concisely.

6.1 Background

SPPs encode transformations on packets in a symbolic manner. Hence, we can construct automata that use SPPs to represent the transition and observation functions for each state.

Definition 6.1. A *symbolic* NetKAT automaton (SNKA) is a four-tuple $(S, s_0, \delta, \varepsilon)$ where S is a finite set of states, $s_0 \in S$ is the start state, δ and ε are transition and observation functions, respectively:

$$\delta\colon S \to S \to \mathsf{SPP} \qquad \varepsilon\colon S \to \mathsf{SPP}$$

There is one additional restriction to ensure that automata are deterministic: for all states $s, s_1, s_2 \in S$ such that $s_1 \neq s_2$, we must have $\delta(s)(s_1) \mathbin{\hat{\cap}} \delta(s)(s_2) = \bot$, where $\hat{\cap}$ is intersection on SPPs.

Semantically, an SNKA $\mathcal{M} = (S, s_0, \delta, \varepsilon)$ accepts a language $\mathcal{L}(\mathcal{M}) \subseteq \mathsf{GS}$ containing all strings $w \in \mathsf{GS}$ for which $\mathcal{M}(s_0, w) = \top$, where:

$$\mathcal{M}(s, \alpha \cdot \beta) \triangleq [\![\varepsilon(s)]\!](\alpha, \beta)$$

$$\mathcal{M}(s, \alpha \cdot \beta \cdot \mathsf{dup} \cdot w') \triangleq \begin{cases} \mathcal{M}(s', \beta \cdot w') & \exists s' \in S \text{ such that } [\![\delta(s)(s')]\!](\alpha, \beta) = \top \\ \bot & \text{otherwise} \end{cases}$$

6.2 The SNKA Teacher

For the SNKA learning problem, we assume essentially the same teacher as in Section 4, except that the teacher supports equivalence queries for symbolic automata instead of PNKA.

Definition 6.2 (MAT Interface). The MAT interface for Symbolic NetKAT automata (SNKA) is implemented as a membership and an equivalence oracle:

$$\mathsf{mem}_{\mathsf{SNKA}} : \mathsf{GS} \to 2 \qquad \mathsf{equiv}_{\mathsf{SNKA}} : \mathsf{SNKA} \to 1 + \mathsf{GS}$$

Remark 2. *The concept of using isolated MAT-style learning subroutines for complex transition labels was first explored by Argyros and D'Antoni [5] in the context of SFAs. However, using SFA learning for NetKAT automata would only handle the transitions in a symbolic way—i.e., the state space would blow up, as with PNKA learning. Hence, to obtain an efficient algorithm for SNKA learning requires additional insights and data structures, as we show below.*

Our general strategy for the learning algorithm is to follow the canonical learning algorithm of Figure 2, but eliminating *all* of the steps where we have to iterate over the entire packet space. To review, here are the steps where the canonical learner (Figure 2) iterates over the packet space:

- We initialize a table T_α for every packet α.
- We initialize every E_α to include every individual packet in Pk.
- In two places we apply $\mathsf{extend}(T)$: Since we may think of T_α as already having $|\mathsf{Pk}|$ rows and columns (i.e., of size $|S_\alpha \cup S_\alpha \cdot (\mathsf{Pk} \cdot \mathsf{dup})|$ and $|E_\alpha|$, respectively), then extend requires at least $|\mathsf{Pk}|$ membership queries because we have just added a row or column.

Instead, we will adopt a kind of lazy approach—e.g., rather than eagerly asking $|\mathsf{Pk}|$ many membership queries, we will make incremental guesses and wait to be corrected by a counterexample. In particular, even if we remove all three of these steps, our conjectures will still apply to the entire packet space due to use of the EPP to SPP conversion (Section 5). Put another way, we build an automaton with the specific evidence traces that we have seen, and then generalize the automaton from there to be well-defined on all input traces. Before specifying our learning algorithm in pseudocode, we first describe the modifications we make to the core data structures.

6.3 Partial Observation Table

We allow the observation table to be partial—i.e., entire packet tables as well as individual entries in packet tables may be missing. Hence, at a given point in the learning process, we will only have a packet tables for the packets we have encountered in some example. In addition, the packet table function T_α does not have information on the extended portion (S'_α), nor is E_α initialized with all of Pk. Putting these ideas together, the type of a partial packet table is as follows:

Definition 6.3 (Partial Packet Table). For a packet $\alpha \in \mathsf{Pk}$, a packet table is a tuple $(S_\alpha, E_\alpha, T_\alpha)$, where:

- $S_\alpha \subseteq \{p \mid \mathsf{last}(p) = \alpha, p \in \mathsf{Pre}\}$ is a set of *access prefixes.*
- $E_\alpha \subseteq \mathsf{Suf}$ is a set of *distinguishing suffixes.*
- $T_\alpha \colon S_\alpha \times E_\alpha \to 2$ is defined such that $T_\alpha(s, e) = \top$ if $s \cdot e \in \mathcal{L}$, and $T_\alpha(s, e) = \bot$ otherwise.

Definition 6.4 (Partial Observation Table). An observation table P is a set of Partial Packet Tables, $(S_\alpha, E_\alpha, T_\alpha)$. There may or may not be a Partial Packet Table in P for each particular $\alpha \in \mathsf{Pk}$.

Closedness and Consistency. Along with these changes to the table structure, we will omit the closedness check in our algorithm. The reason we can avoid this check is that, since we will not be making membership queries for an extension of every prefix by one packet, we will also not have any corresponding "lower" rows of our packet-tables. We will only add transitions for which we have received evidence. Note that missing transitions are perfectly acceptable in a symbolic automaton—the result is simply to drop certain traces.

We will still check a form of consistency as this is needed to ensure that the hypothesis automaton is deterministic (i.e., consistency will ensure we only allow a transition to one state for a packet pair). The modification to consistency is simple: whenever we have equal rows in a partial packet table $(S_\alpha, E_\alpha, T_\alpha)$, we only require that their successors on, say β, are equal *if* both rows exist in $(S_\beta, E_\beta, T_\beta)$ (the box highlights the difference from Definition 4.6).

```
while $P$ not consistent do
    let $e \in E_\beta$ s.t. $\mathrm{row}_\alpha(s) = \mathrm{row}_\alpha(s')$,
      but $T_\beta(s \cdot \beta \cdot \mathrm{dup}, e) \neq T_\beta(s' \cdot \beta \cdot \mathrm{dup}, e)$.
    $E_\alpha \leftarrow E_\alpha \cup \{\beta \cdot \mathrm{dup} \cdot e\}$
return extend$(P)$
```

(a) Algorithm for mkConsistent: $T \to T$.

```
$S \leftarrow \{s_0\}$
$\delta(s_0)(s_0) \leftarrow \bot$
$\varepsilon(s_0) \leftarrow \bot$
return $\mathcal{M}_\varnothing \leftarrow (S, s_0, \delta, \varepsilon)$
```

(b) $\mathcal{M}_\varnothing$ is the SNKA of the empty language.

```
$\mathcal{H} \leftarrow \mathcal{M}_\varnothing$; $P \leftarrow \{\}$
while $c \leftarrow \mathrm{equiv}_{\mathsf{SNKA}}(\mathcal{H}) \in \mathrm{GS}$ do
    $P \leftarrow$ update$(P, c)$
    $P \leftarrow$ mkConsistent$(P)$
    $\mathcal{H} \leftarrow \mathrm{hyp}_{\mathsf{SNKA}}(P)$
return $\mathcal{H}$
```

(c) Algorithm for learning symbolic automata, NKL*.

```
Input: Partial Observation table $P$, counterexample string $c \in \mathrm{GS}$
for $s \in \mathrm{Pre}, e \in \mathrm{Suf}$ s.t. $c = s \cdot e$ do
    $S_{\mathrm{last}(s)} \leftarrow S_{\mathrm{last}(s)} \cup \{s\}$
    $E_{\mathrm{last}(s)} \leftarrow E_{\mathrm{last}(s)} \cup \{e\}$
return extend$(P)$
```

(d) Subroutine update.

Fig. 6. SNKA learning algorithm and subroutines.

Definition 6.5 (Consistent Partial Observation Table). A partial observation table P is *consistent*, if for each $\alpha, \beta \in \mathrm{Pk}$ and for each $s, s' \in S_\alpha$, then $\boxed{\text{if both } s \cdot \beta \cdot \mathrm{dup}, s' \cdot \beta \cdot \mathrm{dup} \in S_\beta}$, and we have $\mathrm{row}_\alpha(s) = \mathrm{row}_\alpha(s')$, then we must have $\mathrm{row}_\beta(s \cdot \beta \cdot \mathrm{dup}) = \mathrm{row}_\beta(s' \cdot \beta \cdot \mathrm{dup})$.

6.4 The SNKA Learning Algorithm

The rest of the SNKA learning algorithm is similar to the one given in Figure 2. The update function we defined before (Figure 2) almost still works except that we add both *prefixes and suffixes* of each counterexample to P. The updated pseudocode is shown in Figure 6.

Due to our dramatic simplification of the outer algorithm, however, the construction of our hypothesis automaton, $\mathrm{hyp}_{\mathsf{SNKA}}$, is more complicated explained in the next section.

6.5 Constructing Symbolic Hypotheses

Given an observation table P that classifies the information observed so far, we now show how to construct an SNKA as a hypothesis. To separate concerns and tame the complexity somewhat, we construct our automaton in two phases. We first use the observation table to build a data structure we call an *evidence automaton* (EA) and then we convert the EA to an SNKA.

6.5.1 Constructing the Evidence Automaton. An EA is like an SNKA but uses EPPs instead of SPPs.

Definition 6.6 (Evidence Automaton). A NetKAT evidence automaton (*EA*) is a four-tuple $(Q, q_0, \delta, \varepsilon)$, with Q a finite set of states, $q_0 \in S$ the initial state, $\delta\colon Q \to Q \to \mathrm{EPP}$ the transition function and $\varepsilon\colon Q \to \mathrm{EPP}$ the observation function.

Now we show how to construct an evidence automaton $\mathcal{E} = (Q, q_0, \delta, \varepsilon)$ from a partial observation table P. The first challenge is to create the set of states for the evidence automaton. Here, we are guided by the semantics of the canonical automaton but wish to have as few states as possible.

Packet-specific states. For each packet table $(S_\alpha, E_\alpha, T_\alpha) \in P$, we start with a set of states by inspecting the packet table T_α the same way that they are identified in Section 4.

So for each α, we define the packet-specific states for each α we have $Q_\alpha \triangleq \{\mathrm{row}_\alpha(s) \mid s \in S_\alpha\}$. This means that for each individual elements $q, q' \in Q_\alpha$ we have prefixes $s, s' \in S$ such that $q = \mathrm{row}_\alpha(s)$ and $q' = \mathrm{row}_\alpha(s')$. Moreover, if $q \neq q'$, this means that there is a suffix $e \in E_\alpha$ for which $\mathrm{row}_\alpha(s)(e) \neq \mathrm{row}_\alpha(s')(e)$, or, equivalently, $T_\alpha(s, e) \neq T_\alpha(s', e)$.

Thus Q_α are just the set of distinct rows of the packet table $(S_\alpha, E_\alpha, T_\alpha)$ that can be distinguished by suffixes from E_α. Next, we merge these states to produce the set of states of the symbolic NKA.

Merged NKA states. Now that we have Q_α for each α, we build a set Q, along with a map for each Q_α into Q, which we denote $\eta\colon Q_\alpha \to Q$. There are only two restrictions that guide how we can combine packet-specific states into global states:

(1) There is a special $q_0 \in Q$ such that for all Q_α, we have $\eta(\mathrm{row}_\alpha\ \alpha) = q_0$.
(2) For each $q, q' \in Q_\alpha$ then $q \neq q'$ implies $\eta(q) \neq \eta(q')$.

The first restriction says that we must map every start state into the same global state. The second restriction says that we cannot combine states from the same Q_α, after all these are already distinguished by a suffix in E_α, and this suffix prevents their combination in the symbolic automaton.

Every combination of packet states into global states that respects these two restrictions is valid. The reason, briefly, is that we may simply guard every behavior by a complete test for each different packet in the worst case (the final SPP might not do this, because it might not be needed). Indeed, different implementations may make different merges here. Different choices for η will result in different sizes for the SPPs in the final automaton (state-minimal SNKAs are not unique).

Although it would be interesting to explore heuristics that result in smaller final SPPs, we defer that question to future work and instead merely insist that the result be minimal with respect to the number of states. To achieve this, we choose Q to have a size matching the largest $|Q_\alpha|$ (clearly any smaller Q will violate restriction 2, above). Otherwise we assume an arbitrary choice for η.

Adding EA Transitions. Having created the set of states, Q, we need to build an EPP to label the transitions between each pair of states to build δ. To do this, we start by initializing an empty EPP for each pair of states. Next we iterate over the entire observation table adding a positive example pair to some EPP for each row in each packet table. That is, for each trace $s \cdot \alpha \cdot \mathrm{dup} \cdot \beta \in S_\beta$, we add a pair (α, β) with label $\top$ to the EPP $\delta(\eta(\mathrm{row}_\alpha(s \cdot \alpha)))(\eta(\mathrm{row}_\beta(s \cdot \alpha \cdot \mathrm{dup} \cdot \beta)))$.

Determinizing pairs. We would like to perform EPP to SPP conversion on each transition and observation function, but there is a fly in the ointment. Because we will convert each transition to SPP independently, two SPPs leaving the same state might generalize to "overlap" such that the resulting automaton is nondeterministic. For example, suppose we have two EPPs labeling two transitions leaving the same state, s, as shown in Figure 7a. If we perform SPP conversion to the EA on the left, we get the symbolic NKA on the right—which is not deterministic! Leaving state s on the packet pair $(\{f \mapsto 1\}, \{f \mapsto 1\})$ we can go to either s' or s'' because $[\![\top]\!](\{f \mapsto 1\}, \{f \mapsto 1\}) = \top$. To prevent this problem, we add negative examples to the EA: for each positive example pair (α, β) such that $[\![\delta(s)(s')]\!](\alpha, \beta) = \top$, we add a negative pair, $(\alpha, \beta, \bot)$ to each EPP $\delta(s)(s'')$ where $s' \neq s''$. This process requires that we do not have two of the same *positive example pairs* leading to different states out of the same state, but this is ensured by the consistency check.

Adding EA Observations. Defining the observation function for an SNKA is analogous to determining whether a state in a standard DFA is accepting or not. Recall that in the canonical learner, we are guaranteed to have $\mathsf{Pk} \subseteq E_\alpha$ which allows us to define the observation function on every final packet for each row. We perform the analogous step here. More precisely, for each $q = \mathrm{row}_\alpha(s)$:

$$[\![\varepsilon(\eta(q))]\!](\alpha, \beta) = \ell \iff T_\alpha(s, \beta) \text{ exists and } T_\alpha(s, \beta) = \ell.$$

To summarize the EA construction: for a partial observation table P, we have $\mathrm{ev}(P) \triangleq (Q, q_0, \delta, \varepsilon)$ constructed by the following pseudocode:

```
Choose $Q$ with special $q_0 \in Q$ such that $|Q| = \max_\alpha |Q_\alpha|$.
Choose $\eta$ subject to restrictions (1) and (2) in Section 6.5.1.
```

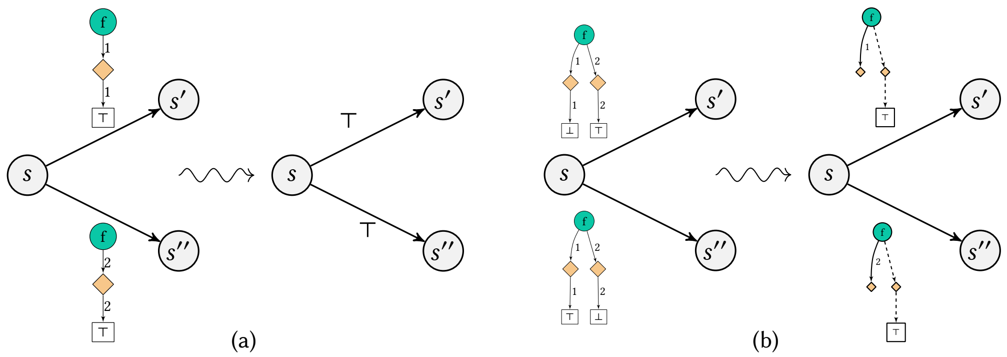


Fig. 7. Converting EPP transition labels to SPP transition labels.

```
for $w \in S_\alpha$ s.t. $w \cdot \beta \cdot \mathrm{dup} \in S_\beta$ do
    Update $\delta(\eta(\mathrm{row}_\alpha\ w))(\eta(\mathrm{row}_\beta(w \cdot \beta \cdot \mathrm{dup})))$ with $(\alpha, \beta), \top$.
for $w \in S_\alpha, \beta \in E_\alpha$ do
    Update $\varepsilon(\eta(\mathrm{row}_\alpha(w)))$ with $(\alpha, \beta), T_\alpha(w, \beta)$.
```

6.6 Converting the Evidence Automaton to Symbolic NetKAT Automaton

Having constructed an EA from the Observation Table, we are close to having an SNKA to conjecture. Conceptually, we would like to simply map our $\mathsf{hyp}_{\mathsf{SPP}}$ function from Section 5 over the transition and observation EPPs in the EA, producing an SNKA. There is one final determinization problem with doing this, which we show by way of an example.

Example 6.7. Suppose we have added the negative example pairs to the EPPs, shown in Figure 7b. Unfortunately, when we do the SPP conversion, we may still get the automaton on the right, which is still not deterministic! The SPPs for $f \neq 1$ and $f \neq 2$ overlap on, e.g., $(\{f \mapsto 3\}, \{f \mapsto 3\})$.

This overlap illustrated by Example 6.7 can be resolved by taking differences arbitrarily (as SPP operations). The reason is that any pairs remaining in the intersection of the semantics of two same-origin SPPs must *both* not be from example pairs in the EPPs (because otherwise the overlap would have been prevented by negative pairs, above). We can therefore compute final transition SPPs by subtracting out the other SPPs leaving the same state.

Thus, given an EA $\mathcal{E} = (Q, q_0, \delta, \varepsilon)$ we define $\mathsf{sym}(\mathcal{E}) \triangleq (Q, q_0, \delta', \varepsilon')$, where [3]:

$$\delta'(q)(q') \triangleq \mathsf{hyp}_{\mathsf{SPP}}(q)(q') \mathbin{\hat{-}} \hat{\sum_{q'' \neq q'}} \mathsf{hyp}_{\mathsf{SPP}}(q)(q'') \qquad \varepsilon'(q) \triangleq \mathsf{hyp}_{\mathsf{SPP}}(\varepsilon(q))$$

6.7 Correctness of the SNKA Learner

We conclude by showing that our symbolic SNKA learner is correct. The first lemma says that each conjecture is a well-defined SNKA.

Lemma 6.8. *For any partial observation table P, we have that $\mathsf{hyp}_{\mathsf{SNKA}}(P)$ is a well-defined SNKA.*

Lemma 6.9. *Let $\mathcal{E} = (Q, q_0, \delta, \varepsilon)$ be an EA for a consistent partial observation table P (i.e. $\mathcal{E} = \mathsf{ev}(P)$), and let $\mathcal{M} = (Q, q_0, \delta', \varepsilon) = \mathsf{sym}(\mathcal{E})$. For any $q, q' \in Q$ and any pair $(\alpha, \beta) \in \mathsf{Pk} \times \mathsf{Pk}$, then:*

(a) If $[\![\delta(q)(q')]\!](\alpha, \beta) = \top$, then $[\![\delta'(q)(q')]\!](\alpha, \beta) = \top$.
(b) If $[\![\varepsilon(q)]\!](\alpha, \beta) = \top$, then $[\![\varepsilon'(q)]\!](\alpha, \beta) = \top$.

[3]Using $\hat{-}$ for semantic difference of SPPs, and $\hat{\sum}$ for sum of SPPs

Next we show that hypothesis automata are correct for certain examples in the table.

Lemma 6.10. *Let P be a consistent partial observation table for target language $\mathcal{L} \subseteq \mathsf{GS}$ and $\mathcal{M} = \mathsf{hyp}_{\mathsf{SNKA}}(P) = (S, s_0, \delta, \varepsilon)$. Let $u \in \mathsf{GS}$. If for every "breaking point" of $u = w \cdot e$ there is a packet table $(S_\alpha, E_\alpha, T_\alpha)$ such that $w \in S_\alpha$ and $e \in E_\alpha$, then we have $\mathcal{M}(\eta(\mathsf{row}_\alpha(w), \alpha \cdot e) = \top \iff T_\alpha(w, e) = \top$.*

Corollary 6.11. *After receiving a counterexample c, any subsequent hypothesis $\mathcal{M} = (S, s_0, \delta, \varepsilon)$ is correct with respect to the target $\mathcal{L}$ on c.*

Theorem 6.12. *For a regular language $\mathcal{L}$, $\mathsf{NKL}^\star$ (in Figure 6c) terminates with an SNKA for $\mathcal{L}$.*

Query complexity. It is possible to model NetKAT using DFAs, where each packet is a character. Using this representation, the standard $\mathsf{L}^\star$ algorithm would learn a model using polynomial many queries in the size of this DFA. However, this representation of a NetKAT program is in general much larger than the SNKA learned by the algorithm in Section 6. Our formal development establishes termination of the SNKA learner, but we leave a more precise analysis to future work. It is important to note that the two algorithms cannot be directly compared because the assumption of equivalence oracle is different for the two models. As the symbolic representations used in SPPs and SNKAs are designed to be compact in the common case, establishing complexity is likely to be more complicated than for $\mathsf{L}^\star$. We note, however, that the PNKA learner (Section 4) inherits polynomial query complexity in the size of the PNKA by essentially the same arguments applicable to $\mathsf{L}^\star$.

6.8 Networking Example

We now return to the example network that we encoded in NetKAT in Section 2.1. As a reminder, the full target expression is:

$$\mathsf{sw}{=}1 \cdot \mathsf{pt}{=}1 \cdot ((\mathsf{pt}{=}1 \cdot \mathsf{pt}{\leftarrow}2 + \mathsf{pt}{=}2 \cdot \mathsf{pt}{\leftarrow}1) \cdot$$
$$(\mathsf{pt}{=}1 + \mathsf{pt}{=}3 + \mathsf{pt}{=}2 \cdot (\mathsf{sw}{=}1 \cdot \mathsf{sw}{\leftarrow}2 + \mathsf{sw}{=}2 \cdot \mathsf{sw}{\leftarrow}1)) \cdot \mathsf{dup})^\star \cdot \mathsf{sw}{=}2 \cdot \mathsf{pt}{=}1$$

Our algorithm succeeds in learning a small automaton for this expression after 6 conjectures and 31 membership queries. For readability, we omit membership queries and observation tables. The SPP labels of transitions are shown using equivalent dup-free expressions.

The learner starts by conjecturing the "drop-everything" automaton, on the right in Figure 8. The teacher responds with a positive example (valid trace):

$$[\{\mathsf{sw} \mapsto 1, \mathsf{pt} \mapsto 1\}; \{\mathsf{sw} \mapsto 2, \mathsf{pt} \mapsto 2\}; \{\mathsf{sw} \mapsto 2, \mathsf{pt} \mapsto 1\}; \{\mathsf{sw} \mapsto 2, \mathsf{pt} \mapsto 1\}]$$

And then the learner conjectures the automaton in the middle in Figure 8.

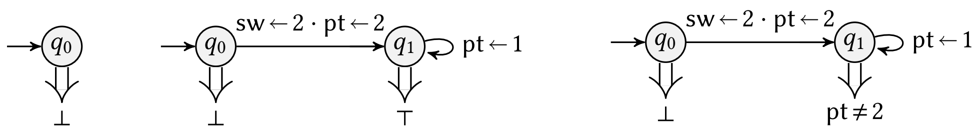


Fig. 8. The first three conjectures.

The teacher then provides a negative example (an invalid trace):

$$[\{\mathsf{sw} \mapsto 1, \mathsf{pt} \mapsto 1\}; \{\mathsf{sw} \mapsto 2, \mathsf{pt} \mapsto 2\}; \{\mathsf{sw} \mapsto 2, \mathsf{pt} \mapsto 2\}]$$

and the learner conjectures the automaton on the right in Figure 8. The process continues with the teacher giving another negative example:

$$[\{\mathsf{sw} \mapsto 1, \mathsf{pt} \mapsto 1\}; \{\mathsf{sw} \mapsto 2, \mathsf{pt} \mapsto 2\}; \{\mathsf{sw} \mapsto 2, \mathsf{pt} \mapsto 1\}; \{\mathsf{sw} \mapsto 2, \mathsf{pt} \mapsto 1\}; \{\mathsf{sw} \mapsto 2, \mathsf{pt} \mapsto 1\}]$$

and the learner conjecturing the automaton on the left in Figure 9. The teacher gives yet another

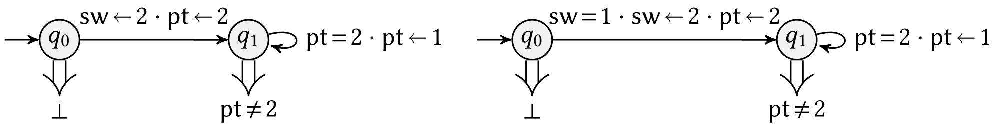


Fig. 9. Two further conjectures.

negative example— $[\{\text{sw} \mapsto 0, \text{pt} \mapsto 0\}; \{\text{sw} \mapsto 2, \text{pt} \mapsto 2\}; \{\text{sw} \mapsto 2, \text{pt} \mapsto 1\}; \{\text{sw} \mapsto 2, \text{pt} \mapsto 1\}]$— followed by the learner conjecturing the automaton on the right in Figure 9.

Finally, the teacher gives the negative example:

$$[\{\text{sw} \mapsto 1, \text{pt} \mapsto 0\}; \{\text{sw} \mapsto 2, \text{pt} \mapsto 2\}; \{\text{sw} \mapsto 2, \text{pt} \mapsto 1\}; \{\text{sw} \mapsto 2, \text{pt} \mapsto 1\}]$$

And the learner conjectures:

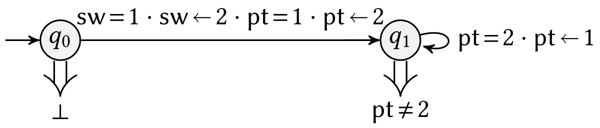


Fig. 10. The sixth (and last) conjecture.

This automaton is correct and the learning process terminates successfully! Though this is just a toy example, meant to illustrate the steps of the algorithm, it is worth noting that the larger networks that we deal with in the next section follow a similar structure.

7 IMPLEMENTATION AND EVALUATION

To evaluate our techniques, we have prototyped our algorithms for learning SPPs (Figure 4) and SNKAs (Section 6) in OCaml. Although our implementation has not yet been tuned to achieve scalable performance, it is still useful for illustrating the benefits of our approach. We have used it to experiment with learning models of real-world networks drawn from a standard benchmark.

7.1 Evaluation

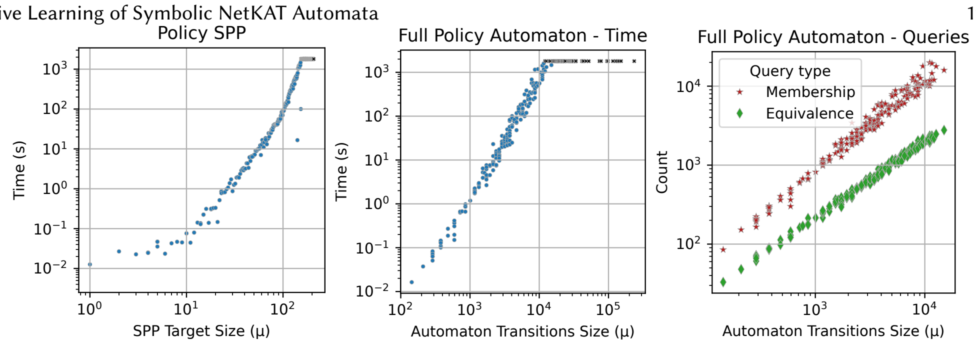
To evaluate our approach we sought to answer two research questions:

- How well does the SPP learner scale with topology size for network transfer functions?
- How does the SNKA learner scale with topology size for whole-trace network behavior?

Topology Zoo Policies. To explore these questions, we built a set of learning problem instances using the Internet Topology Zoo [21], a dataset for benchmarking network verification tools [16, 27]. For each topology, we have a simple shortest-path routing policy encoded as in Section 2.1.

SPP Learning. To evaluate the SPP learner, we set each network's "transfer function" ($r \cdot t$, see Section 2.1) as the target SPP and ran our SPP learner (that is, membership queries are on pairs of packets representing single-step transformations in the network). The results are shown in Figure 11a in log-log scales. The learner recovered the correct SPP for 190 out of 261 topologies before reaching a 30 minutes timeout, suggesting its feasibility for learning policies of modest size.

SNKA Learning. To evaluate the SNKA learner, we instantiated the teacher with a full network behavior expression, $p_i \cdot (r \cdot t \cdot \text{dup})^\star \cdot p_f$. For our input and output predicates, we choose: $p_i \triangleq \sum_i \text{sw}=i \cdot (\sum_j \text{dst}=j)$, and $p_f \triangleq \sum_i \text{sw}=i \cdot \text{dst}=i$. This says that we are interested in initial packets starting at any origin with any destination, and the final packets which have reached their destination (i.e., membership queries are full-trace behaviors of the policy). The timing results are shown in Figure 11b. The SNKA learner succeeded on learning 190 of the 261 topologies. A plot of



(a) SPP learner timing. (b) SNKA learner timing. (c) SNKA learner query counts.

Fig. 11. Benchmark results for Topology Zoo dataset. Examples that exceeded 30 minutes are depicted with an "x" in the timing plot; the queries plot includes only examples that completed within the time limit. To keep the plot legible, we omit the largest topology, KDL, as it is an outlier.

the number of membership queries and equivalence versus target size is shown in Figure 11c. The linear shape does not mean linear behavior because the axes are log-log.

Remark 3. *The largest topology solved by the SNKA learner, Uunet, has 48 nodes, with 14 port numbers. Considering the fields* sw, dst, *and* pt *used in the problem, we have* $|\mathsf{Pk}| = 48 \cdot 48 \cdot 14 = 32256$. *Because of the carry-on packet semantics, this means that any approach based on learning a DFA would need to identify many thousands of different* states, *even if the transitions are treated symbolically, since only carry-on packets that will be dropped can be combined. Instead, the SNKA learner here identifies an automaton with only 2 states (excluding a drop state), and even though the transition SPPs are complex, it does so after* only *15,932 membership queries, and 2775 equivalence queries.*

8 PRACTICAL CONCERNS

In this section we sketch additional tasks that would be needed to apply our techniques in a practical setting where the oracle is a closed-box network or network component.

8.1 Membership queries

We assume a membership oracle of type $\mathsf{GS} \to 2$ because it allows us to apply the core of MAT style learning to our domain. One might observe, however, that a closed-box network might more naturally be modeled by a type like $\mathsf{Pk} \to \mathsf{GS}$ (i.e., given input packets, traces are produced), and a real system may have nondeterministic behavior. To run our algorithm on a closed-box system, one would need to implement this translation between input packets and input traces. This is not a big issue: we can approximate the $\mathsf{GS} \to 2$ oracle using a "real" oracle by sending the first packet of our desired query trace (perhaps multiple times) and then checking whether the query trace is produced by the system in any of the trials (caching the results for later queries).

8.2 Equivalence queries

The assumption of an equivalence oracle in work based on Angluin's $\mathsf{L}^\star$ algorithm, including ours, is indeed a strong assumption. However, in practice, it often suffices to approximate equivalence. There is a large body of prior work (e.g., CacheQuery from PLDI '20 [31]), showing that approximation is effective, so the assumption of an equivalence oracle is less of a practical hurdle than it might seem.

In many systems, equivalence queries are approximated using membership queries. In a nutshell, the idea is to first generate a large set of traces (e.g., all traces up to a given length, or a random sample of such traces) and then execute them on the conjectured model and the system. If no discrepancies are found, then the equivalence query is deemed to be successful. On the other hand, if a discrepancy is found, then the equivalence query is deemed to fail, and the trace that triggered the discrepancy is returned as a counter-example. The Random Wp-Method [17] and domain-specific approaches [23] offer reasonable approximation guarantees.

9 RELATED WORK

This paper is the first to address symbolic learning of NetKAT automata, however, a number of similar problems have been the topic of extensive study.

Learning for GKAT. Zetzsche et al. [33] adapt Angluin's $\mathsf{L}^\star$ to the setting of Guarded Kleene Algebra with Tests. Their work bears some similarity to ours in that they also establish a Myhill-Nerode Theorem for GKAT en route to the full presentation of their algorithm. Unfortunately, we cannot apply their results beyond inspiration because of the differences between GKAT and NetKAT: GKAT supports only a "guarded" choice operator, which is deterministic, while NetKAT supports the full nondeterministic choice operation of KAT. Further, NetKAT's carry-on packet semantics is the source of the challenges that we solve in our paper, issues that have no analog in GKAT. The incompatibility of NetKAT and GKAT is explored in Wasserstein's Master's thesis [32].

Learning OBDDs. Angluin-style learning of Ordered Binary Decision Diagrams (OBDDs) [18] and of automata with decision diagram transitions [24] have been explored. BDDs also appear in the use of $\mathsf{L}^\star$ to learn Java interface specifications [2] . The structure we use, Symbolic Packet Programs, are reminiscent of decision diagrams, yet have crucial differences which are essential to encode NetKAT policies as discussed in [27], which require the design of a new learning algorithm.

Learning Networking Protocols. Fiterău-Broştean et al. [13] successfully applied automata learning to learn the state machine of fragments of the TCP protocol, and identified bugs in a widely used TCP implementation. This work was further extended to learning sliding window behavior [12], learning SSH implementations [15], and DTLS implementations [14]. Ferreira et al. [10] introduced the Prognosis tool to apply automata learning to the QUIC protocol, a promising new network protocol aiming to fully replace TCP, TLS/SSL, and HTTP for the modern web. Prognosis was used to identify issues with both the QUIC RFC itself and several implementations. These approaches are prime examples of using automata learning to identify bugs in networking, but they have mostly explored DFAs, which have the drawbacks mentioned above. Our paper brings NetKAT automata as a more expressive learning target to the learning toolkit for networking.

10 CONCLUSION

In this paper we presented novel techniques for active learning of NetKAT models. We developed the theory of NetKAT with a characterization of canonical automata (akin to the results of Myhill-Nerode for classical automata) and then used this notion to develop a learning algorithm. We designed two symbolic learning algorithms, one for the dup-free fragment of NetKAT (SPP learner) and one for the general case of symbolic NetKAT automata (SNKA learner). We implemented these algorithms in an OCaml prototype and performed an evaluation using existing benchmarks. Our evaluation shows that both the SPP and the SNKA learners scale with topology size.

As a natural next step, we would like to apply our algorithm to practical scenarios, including inferring the model of a device for which no configuration or program is known and analyze potential misconfigurations or malicious behavior. We would also like to explore further improvements to the algorithm via *symbolic packet tables*. Currently, if multiple packets are observed to behave equivalently, we still treat their packet tables independently. This leads to repetition in queries, and a higher memory footprint than potentially needed. We would also like to investigate efficient heuristics for implementations of the state grouping function η. Currently the properties of this function focus on folding the state space, however transition SPP minimality would be as important for the scalability of NetKAT procedures that use the models.

ACKNOWLEDGEMENTS

We are grateful to our PLDI reviewers and shepherd who helped us improve our paper significantly. We also thank David Darais, Caleb Lucas-Foley, and Cole Schlesinger of Galois Inc. for helpful conversations on our early progress and to the Cornell PLDG for their feedback. We are also grateful 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 Autoprobe (no. 101002697), and a Royal Society Wolfson fellowship, as well as gifts from Google, InfoSys, and the VMware University Research Fund.

DATA AVAILABILITY

A snapshot of the OCaml code which underwent artifact evaluation is available as a dependencies-included Docker image on Zenodo [26].

REFERENCES

[1] Kinan Dak Albab, Jonathan DiLorenzo, Stefan Heule, Ali Kheradmand, Steffen Smolka, Konstantin Weitz, Muhammad Timarzi, Jiaqi Gao, and Minlan Yu. 2022. SwitchV: automated SDN switch validation with P4 models. In *Proceedings of the ACM SIGCOMM 2022 Conference* (Amsterdam, Netherlands) *(SIGCOMM 2022)*. Association for Computing Machinery, New York, NY, USA, 365–379. https://doi.org/10.1145/3544216.3544220

[2] Rajeev Alur, Pavol Černý, P. Madhusudan, and Wonhong Nam. 2005. Synthesis of Interface Specifications for Java Classes. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Long Beach, California, USA) *(POPL 2005)*. Association for Computing Machinery, New York, NY, USA, 98–109. https://doi.org/10.1145/1040305.1040314

[3] Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. 2014. NetKAT: Semantic Foundations for Networks. In *Proceedings of the 41st ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2014)*. https://doi.org/10.1145/2535838.2535862

[4] Dana Angluin. 1987. Learning regular sets from queries and counterexamples. *Information and Computation* 75, 2 (1987), 87–106. https://doi.org/10.1016/0890-5401(87)90052-6

[5] George Argyros and Loris D'Antoni. 2018. The Learnability of Symbolic Automata. In *Computer Aided Verification*, Hana Chockler and Georg Weissenbacher (Eds.). Springer International Publishing, Cham, 427–445. https://doi.org/10.1007/978-3-319-96145-3_23

[6] Benedikt Bollig, Peter Habermehl, Carsten Kern, and Martin Leucker. 2009. Angluin-style learning of NFA. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence* (Pasadena, California, USA) *(IJCAI'09)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1004–1009. http://ijcai.org/papers09/Papers/IJCAI09-170.pdf

[7] Sofia Cassel, Falk Howar, Bengt Jonsson, and Bernhard Steffen. 2014. Learning Extended Finite State Machines. In *Software Engineering and Formal Methods*, Dimitra Giannakopoulou and Gwen Salaün (Eds.). Springer International Publishing, Cham, 250–264. https://doi.org/10.1007/978-3-319-10431-7_18

[8] Loris D'Antoni and Margus Veanes. 2017. The Power of Symbolic Automata and Transducers. In *Computer Aided Verification*. Springer International Publishing, Cham, 47–67.

[9] Samuel Drews and Loris D'Antoni. 2017. Learning Symbolic Automata. In *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 10205)*, Axel Legay and Tiziana Margaria (Eds.). 173–189. https://doi.org/10.1007/978-3-662-54577-5_10

[10] Tiago Ferreira, Harrison Brewton, Loris D'Antoni, and Alexandra Silva. 2021. Prognosis: closed-box analysis of network protocol implementations. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference* (Virtual Event, USA) *(SIGCOMM 2021)*. Association for Computing Machinery, New York, NY, USA, 762–774. https://doi.org/10.1145/3452296.3472938

[11] Dana Fisman, Hadar Frenkel, and Sandra Zilles. 2023. Inferring Symbolic Automata. Volume 19, Issue 2 (2023), 8899. https://doi.org/10.46298/lmcs-19(2:5)2023

[12] Paul Fiterău-Broştean and Falk Howar. 2017. Learning-Based Testing the Sliding Window Behavior of TCP Implementations. In *Critical Systems: Formal Methods and Automated Verification*, Laure Petrucci, Cristina Seceleanu, and Ana Cavalcanti (Eds.). Springer International Publishing, Cham, 185–200. https://doi.org/10.1007/978-3-319-67113-0_12

[13] Paul Fiterău-Broştean, Ramon Janssen, and Frits Vaandrager. 2014. Learning Fragments of the TCP Network Protocol. In *Formal Methods for Industrial Critical Systems*, Frédéric Lang and Francesco Flammini (Eds.). Springer International Publishing, Cham, 78–93. https://doi.org/10.1007/978-3-319-10702-8_6

[14] Paul Fiterau-Brostean, Bengt Jonsson, Robert Merget, Joeri de Ruiter, Konstantinos Sagonas, and Juraj Somorovsky. 2020. Analysis of DTLS Implementations Using Protocol State Fuzzing. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 2523–2540. https://www.usenix.org/conference/usenixsecurity20/presentation/fiterau-brostean

[15] Paul Fiterau-Brostean, Toon Lenaerts, Erik Poll, Joeri de Ruiter, Frits W. Vaandrager, and Patrick Verleg. 2017. Model learning and model checking of SSH implementations. *Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software* (2017). https://doi.org/10.1145/3092282.3092289

[16] Nate Foster, Dexter Kozen, Mae Milano, Alexandra Silva, and Laure Thompson. 2015. A Coalgebraic Decision Procedure for NetKAT. In *Proceedings of the 42nd ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2015)*. https://doi.org/10.1145/2676726.2677011

[17] S. Fujiwara, G. v. Bochmann, F. Khendek, M. Amalou, and A. Ghedamsi. 1991. Test selection based on finite state models. *IEEE Transactions on Software Engineering* 17, 6 (1991), 591–603. https://doi.org/10.1109/32.87284

[18] Ricard Gavaldà and David Guijarro. 1995. Learning Ordered Binary Decision Diagrams. In *Proceedings of the 6th International Conference on Algorithmic Learning Theory (ALT 1995)*. Springer-Verlag, Berlin, Heidelberg, 228–238. https://doi.org/10.1007/3-540-60454-5_41

[19] Malte Isberner, Falk Howar, and Bernhard Steffen. 2014. The TTT Algorithm: A Redundancy-Free Approach to Active Automata Learning. In *Runtime Verification*, Borzoo Bonakdarpour and Scott A. Smolka (Eds.). Vol. 8734. Springer International Publishing, Cham, 307–322. https://doi.org/10.1007/978-3-319-11164-3_26 Series Title: Lecture Notes in Computer Science.

[20] Michael J. Kearns and Umesh V. Vazirani. 1994. *An Introduction to Computational Learning Theory*. MIT Press, Cambridge, MA, USA. https://doi.org/10.7551/mitpress/3897.001.0001

[21] 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

[22] Dexter Kozen. 2001. *Automata on Guarded Strings and Applications.* Technical Report. USA.

[23] Loes Kruger, Sebastian Junges, and Jurriaan Rot. 2024. Small Test Suites for Active Automata Learning. In *Tools and Algorithms for the Construction and Analysis of Systems: 30th International Conference, TACAS 2024, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2024, Luxembourg City, Luxembourg, April 6–11, 2024, Proceedings, Part II* (Luxembourg City, Luxembourg). Springer-Verlag, Berlin, Heidelberg, 109–129. https://doi.org/10.1007/978-3-031-57249-4_6

[24] Oded Maler and Irini-Eleftheria Mens. 2017. *A Generic Algorithm for Learning Symbolic Automata from Membership Queries.* 146–169. https://doi.org/10.1007/978-3-319-63121-9_8

[25] Mark Moeller, Tiago Ferreira, Thomas Lu, Nate Foster, and Alexandra Silva. 2025. Active Learning of Symbolic NetKAT Automata. arXiv:2504.13794 [cs.PL] https://arxiv.org/pdf/2504.13794

[26] Mark Moeller, Tiago Ferreira, Thomas Lu, Nate Foster, and Alexandra Silva. 2025. *Active Learning of Symbolic NetKAT Automata (Artifact).* https://doi.org/10.5281/zenodo.15230071

[27] 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. In *Proceedings of the 45th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Copenhagen, DK) *(PLDI 2024)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3656454

[28] Anil Nerode. 1958. Linear automaton transformations. https://doi.org/10.1090/S0002-9939-1958-0135681-9

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

[30] Margus Veanes. 2013. Applications of Symbolic Finite Automata. In *Implementation and Application of Automata*, Stavros Konstantinidis (Ed.). Vol. 7982. Springer Berlin Heidelberg, 16–23. https://doi.org/10.1007/978-3-642-39274-0_3 Series Title: Lecture Notes in Computer Science.

[31] Pepe Vila, Pierre Ganty, Marco Guarnieri, and Boris Köpf. 2020. CacheQuery: learning replacement policies from hardware caches. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 519–532. https://doi.org/10.1145/3385412.3386008

[32] Jacob Wasserstein. 2023. Guarded NetKAT: Soundness, Partial-Completeness, Decidability. (May 2023). https://doi.org/10.7298/Y5X5-JR17 Publisher: Cornell University Library.

[33] Stefan Zetzsche, Alexandra Silva, and Matteo Sammartino. 2022. Guarded Kleene Algebra with Tests: Automata Learning. *Electronic Notes in Theoretical Informatics and Computer Science* (2022). https://doi.org/10.46298/entics.10505

Received 2024-11-15; accepted 2025-03-06