Scalable Verification of Probabilistic Networks

Steffen Smolka
Cornell University
Ithaca, NY, USA

Praveen Kumar
Cornell University
Ithaca, NY, USA

David M. Kahn[*]
Carnegie Mellon University
Pittsburgh, PA, USA

Nate Foster
Cornell University
Ithaca, NY, USA

Justin Hsu[*]
University of Wisconsin
Madison, WI, USA

Dexter Kozen
Cornell University
Ithaca, NY, USA

Alexandra Silva
University College London
London, UK

Abstract

This paper presents McNetKAT, a scalable tool for verifying probabilistic network programs. McNetKAT is based on a new semantics for the guarded and history-free fragment of Probabilistic NetKAT in terms of finite-state, absorbing Markov chains. This view allows the semantics of all programs to be computed exactly, enabling construction of an automatic verification tool. Domain-specific optimizations and a parallelizing backend enable McNetKAT to analyze networks with thousands of nodes, automatically reasoning about general properties such as probabilistic program equivalence and refinement, as well as networking properties such as resilience to failures. We evaluate McNetKAT's scalability using real-world topologies, compare its performance against state-of-the-art tools, and develop an extended case study on a recently proposed data center network design.

CCS Concepts • **Theory of computation** → **Automated reasoning**; **Program semantics**; Random walks and Markov chains; • **Networks** → *Network properties*; • **Software and its engineering** → *Domain specific languages*.

Keywords Network verification, Probabilistic Programming

ACM Reference Format:
Steffen Smolka, Praveen Kumar, David M. Kahn, Nate Foster, Justin Hsu, Dexter Kozen, and Alexandra Silva. 2019. Scalable Verification of Probabilistic Networks. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '19), June 22–26, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3314221.3314639

[*]Work performed at Cornell University.

1 Introduction

Networks are among the most complex and critical computing systems used today. Researchers have long sought to develop automated techniques for modeling and analyzing network behavior [40], but only over the last decade has programming language methodology been brought to bear on the problem [5, 6, 28], opening up new avenues for reasoning about networks in a rigorous and principled way [3, 12, 19, 21, 25]. Building on these initial advances, researchers have begun to target more sophisticated networks that exhibit richer phenomena. In particular, there is renewed interest in *randomization* as a tool for designing protocols and modeling behaviors that arise in large-scale systems—from uncertainty about the inputs, to expected load, to likelihood of device and link failures.

Although programming languages for describing randomized networks exist [11, 15], support for automated reasoning remains limited. Even basic properties require quantitative reasoning in the probabilistic setting, and seemingly simple programs can generate complex distributions. Whereas state-of-the-art tools can easily handle deterministic networks with hundreds of thousands of nodes, probabilistic tools are currently orders of magnitude behind.

This paper presents McNetKAT, a new tool for reasoning about probabilistic network programs written in the guarded and history-free fragment of Probabilistic NetKAT (ProbNetKAT) [3, 11, 12, 35]. ProbNetKAT is an expressive programming language based on Kleene Algebra with Tests, capable of modeling a variety of probabilistic behaviors and properties including randomized routing [22, 38], uncertainty about demands [30], and failures [17]. The history-free fragment restricts the language semantics to input-output behavior rather than tracking paths, and the guarded fragment provides conditionals and while loops rather than union and iteration operators. Although the fragment we consider is a restriction of the full language, it is still expressive enough to encode a wide range of practical networking models. Existing deterministic tools, such as Anteater [27], HSA [19], and Veriflow [21], also use guarded and history-free models.

To enable automated reasoning, we first reformulate the semantics of ProbNetKAT in terms of finite state Markov chains. We introduce a *big-step* semantics that models programs as Markov chains that transition from input to output in a single step, using an auxiliary *small-step* semantics to compute the closed-form solution for the semantics of the iteration operator. We prove that the Markov chain semantics coincides with the domain-theoretic semantics for ProbNetKAT developed in previous work [11, 35]. Our new semantics also has a key benefit: the limiting distribution of the resulting Markov chains can be computed exactly in closed form, yielding a concise representation that can be used as the basis for building a practical tool.

We have implemented McNetKAT in an OCaml prototype that takes a ProbNetKAT program as input and produces a stochastic matrix that models its semantics in a finite and explicit form. McNetKAT uses the UMFPACK linear algebra library as a back-end solver to efficiently compute limiting distributions [7], and exploits algebraic properties to automatically parallelize the computation across multiple machines. To facilitate comparisons with other tools, we also developed a back-end based on PRISM [23].

To evaluate the scalability of McNetKAT, we conducted experiments on realistic topologies, routing schemes, and properties. Our results show that McNetKAT scales to networks with thousands of switches, and performs orders of magnitude better than a state-of-the-art tool based on general-purpose symbolic inference [15, 16]. We also used McNetKAT to carry out a case study of the resilience of a fault-tolerant data center design proposed by Liu et al. [26].

Contributions and outline. The central contribution of this paper is the development of a *scalable probabilistic network verification tool*. We develop a new, tractable semantics that is sound with respect to ProbNetKAT's original denotational model. We present a prototype implementation and evaluate it on a variety of scenarios drawn from real-world networks. In §2, we introduce ProbNetKAT using a running example. In §3, we present a semantics based on *finite stochastic matrices* and show that it fully characterizes the behavior of ProbNetKAT programs (Theorem 3.1). In §4, we show how to compute the matrix associated with iteration in closed form. In §5, we discuss our implementation, including symbolic data structures and optimizations that are needed to handle the large state space efficiently. In §6, we evaluate the scalability of McNetKAT on a common data center design and compare its performance against state-of-the-art probabilistic tools. In §7, we present a case study using McNetKAT to analyze resilience in the presence of link failures. We survey related work in §8 and conclude in §9. Proofs can be found in the extended version of this paper [36].

2 Overview

This section introduces a running example that illustrates the main features of the ProbNetKAT language as well as some quantitative network properties that arise in practice.

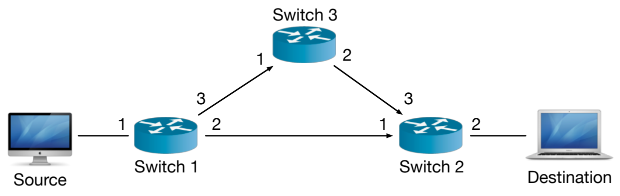


Figure 1. Network topology for running example.

Background on ProbNetKAT. Consider the network in Figure 1, which connects a source to a destination in a topology with three switches. We will first introduce a program that forwards packets from the source to the destination, and then verify that it correctly implements the desired behavior. Next, we will show how to enrich our program to model the possibility of link failures, and develop a fault-tolerant forwarding scheme that automatically routes around failures. Using a quantitative version of program refinement, we will show that the fault-tolerant program is indeed more resilient than the initial program. Finally, we will show how to compute the expected degree of resilience analytically.

To a first approximation, a ProbNetKAT program can be thought of as a randomized function that maps input packets to sets of output packets. Packets are modeled as records, with fields for standard headers—such as the source (`src`) and destination (`dst`) addresses—as well as two fields switch (`sw`) and port (`pt`) encoding the current location of the packet. ProbNetKAT provides several primitives for manipulating packets: a *modification* $f \leftarrow n$ returns the input packet with field f updated to n, while a *test* $f = n$ returns either the input packet unmodified if the test succeeds, or the empty set if the test fails. The primitives skip and drop behave like a test that always succeeds and fails, respectively. In the guarded fragment of the language, programs can be composed sequentially ($p\,;q$), using conditionals (**if** p **then** q_1 **else** q_2), while loops (**while** p **do** q), or probabilistic choice ($p \oplus q$).

Although ProbNetKAT programs can be freely constructed by composing primitive operations, a typical network model is expressed using two programs: a *forwarding* program (sometimes called a *policy*) and a *link* program (sometimes called a *topology*). The forwarding program describes how packets are transformed locally by the switches at each hop. In our running example, to route packets from the source to the destination, switches 1 and 2 can simply forward all incoming packets out on port 2 by modifying the port field (`pt`). This program can be encoded in ProbNetKAT by performing a case analysis on the location of the input packet, and then setting the port field to 2:

$$\begin{aligned} p \triangleq\ & \textbf{if } \texttt{sw}{=}1 \textbf{ then } \texttt{pt}{\leftarrow}2 \textbf{ else} \\ & \textbf{if } \texttt{sw}{=}2 \textbf{ then } \texttt{pt}{\leftarrow}2 \textbf{ else } \text{drop} \end{aligned}$$

The final drop at the end of this program encodes the policy for switch 3, which is unreachable.

We can model the topology as a cascade of conditionals that match packets at the end of each link and update their locations to the link's destination:

$$t \triangleq \textbf{if } \texttt{sw=1}\,;\texttt{pt=2} \textbf{ then } \texttt{sw}\leftarrow\texttt{2}\,;\texttt{pt}\leftarrow\texttt{1} \textbf{ else } \ldots$$

To build the overall network model, we first define predicates for the ingress and egress locations,

$$in \triangleq \texttt{sw=1}\,;\texttt{pt=1} \qquad out \triangleq \texttt{sw=2}\,;\texttt{pt=2}$$

and then combine the forwarding policy p with the topology t. More specifically, a packet traversing the network starts at an ingress and is repeatedly processed by switches and links until it reaches an egress:

$$\mathrm{M}(p,t) \triangleq in\,;p\,;\textbf{while } \neg out \textbf{ do } (t\,;p)$$

We can now state and prove properties about the network by reasoning about this model. For instance, the following equivalence states that p forwards all packets to the destination:

$$\mathrm{M}(p,t) \;\equiv\; in\,;\texttt{sw}\leftarrow\texttt{2}\,;\texttt{pt}\leftarrow\texttt{2}$$

The program on the right can be regarded as an ideal specification that "teleports" each packet to its destination. Such equations were also used in previous work to reason about properties such as waypointing, reachability, isolation, and loop freedom [3, 12].

Probabilistic reasoning. Real-world networks often exhibit nondeterministic behaviors such as fault tolerant routing schemes to handle unexpected failures [26] and randomized algorithms to balance load across multiple paths [22]. Verifying that networks behave as expected in these more complicated scenarios requires a form of probabilistic reasoning, but most state-of-the-art network verification tools model only deterministic behaviors [12, 19, 21].

To illustrate, suppose we want to extend our example with link failures. Most modern switches execute low-level protocols such as Bidirectional Forwarding Detection (BFD) that compute real-time health information about the link connected to each physical port [4]. We can enrich our model so that each switch has a boolean flag $\texttt{up}_i$ that indicates whether the link connected to the switch at port i is up. Then, we can adjust the forwarding logic to use backup paths when the link is down: for switch 1,

$$\begin{aligned}\widehat{p_1} \triangleq\; & \textbf{if } \texttt{up}_2\texttt{=1} \textbf{ then } \texttt{pt}\leftarrow\texttt{2} \textbf{ else} \\ & \textbf{if } \texttt{up}_2\texttt{=0} \textbf{ then } \texttt{pt}\leftarrow\texttt{3} \textbf{ else } \texttt{drop}\end{aligned}$$

and similarly for switches 2 and 3. As before, we can package the forwarding logic for all switches into a single program:

$$\widehat{p} \triangleq \textbf{if } \texttt{sw=1} \textbf{ then } \widehat{p_1} \textbf{ else if } \texttt{sw=2} \textbf{ then } \widehat{p_2} \textbf{ else } \widehat{p_3}$$

Next, we update the encoding of our topology to faithfully model link failures. Links can fail for a wide variety of reasons, including human errors, fiber cuts, and hardware faults. A natural way to model such failures is with a *probabilistic model*—i.e., with a distribution that captures how often certain links fail:

$$\begin{aligned}f_0 &\triangleq \texttt{up}_2\leftarrow\texttt{1}\,;\texttt{up}_3\leftarrow\texttt{1}\\ f_1 &\triangleq \oplus\left\{f_0 \;@\; \tfrac{1}{2}, (\texttt{up}_2\leftarrow\texttt{0}\,;\texttt{up}_3\leftarrow\texttt{1}) \;@\; \tfrac{1}{4}, (\texttt{up}_2\leftarrow\texttt{1}\,;\texttt{up}_3\leftarrow\texttt{0}) \;@\; \tfrac{1}{4}\right\}\\ f_2 &\triangleq (\texttt{up}_2\leftarrow\texttt{1} \oplus_{.8} \texttt{up}_2\leftarrow\texttt{0})\,;(\texttt{up}_3\leftarrow\texttt{1} \oplus_{.8} \texttt{up}_3\leftarrow\texttt{0})\end{aligned}$$

Intuitively, in f_0 no links fail, in f_1 the links ℓ_{12} and ℓ_{13} fail with probability 25% but at most one link fails, while in f_2 the links fail independently with probability 20%. Using the up flags, we can model a topology with possibly faulty links like so:

$$\widehat{t} \triangleq \textbf{if } \texttt{sw=1}\,;\texttt{pt=2}\,;\texttt{up}_2\texttt{=1} \textbf{ then } \texttt{sw}\leftarrow\texttt{2}\,;\texttt{pt}\leftarrow\texttt{1} \textbf{ else } \ldots$$

Combining the policy, topology, and failure model yields a model of the entire network:

$$\begin{aligned}\widehat{\mathrm{M}}(p,t,f) \triangleq\; & \textbf{var } \texttt{up}_2\leftarrow\texttt{1} \textbf{ in}\\ & \textbf{var } \texttt{up}_3\leftarrow\texttt{1} \textbf{ in}\\ & \mathrm{M}((f\,;p),t)\end{aligned}$$

This refined model $\widehat{\mathrm{M}}$ wraps our previous model M with declarations of the two local fields $\texttt{up}_2$ and $\texttt{up}_3$ and executes the failure model (f) at each hop before executing the programs for the switch (p) and topology (t).

Now we can analyze our resilient routing scheme $\widehat{p}$. As a sanity check, we can verify that it delivers packets to their destinations in the absence of failures. Formally, it behaves like the program that teleports packets to their destinations:

$$\widehat{\mathrm{M}}(\widehat{p},\widehat{t},f_0) \;\equiv\; in\,;\texttt{sw}\leftarrow\texttt{2}\,;\texttt{pt}\leftarrow\texttt{2}$$

More interestingly, $\widehat{p}$ is 1-resilient—i.e., it delivers packets provided at most one link fails. Note that this property does *not* hold for the original, naive routing scheme p:

$$\widehat{\mathrm{M}}(\widehat{p},\widehat{t},f_1) \;\equiv\; in\,;\texttt{sw}\leftarrow\texttt{2}\,;\texttt{pt}\leftarrow\texttt{2} \;\not\equiv\; \widehat{\mathrm{M}}(p,\widehat{t},f_1)$$

While $\widehat{p}$ is not fully resilient under failure model f_2, which allows two links to fail simultaneously, we can still show that the refined routing scheme $\widehat{p}$ performs strictly better than the naive scheme p by checking

$$\widehat{\mathrm{M}}(p,\widehat{t},f_2) \;<\; \widehat{\mathrm{M}}(\widehat{p},\widehat{t},f_2)$$

where $p < q$ intuitively means that q delivers packets with higher probability than p.

Going a step further, we might want to compute more general quantitative properties of the distributions generated for a given program. For example, we might compute the probability that each routing scheme delivers packets to the destination under f_2 (i.e., 80% for the naive scheme and 96% for the resilient scheme), potentially valuable information to help an Internet Service Provider (ISP) evaluate a network design to check that it meets certain service-level agreements (SLAs). With this motivation in mind, we aim to build a scalable tool that can carry out automated reasoning on probabilistic network programs expressed in ProbNetKAT.

3 ProbNetKAT Syntax and Semantics

This section reviews the syntax of ProbNetKAT and presents a new semantics based on finite state Markov chains.

Preliminaries. A *packet* π is a record mapping a finite set of fields $f_1, f_2, \ldots, f_k$ to bounded integers n. As we saw in the previous section, fields can include standard header fields such as source (`src`) and destination (`dst`) addresses, as well as logical fields for modeling the current location of the packet in the network or variables such as $\mathtt{up}_i$. These logical fields are not present in a physical network packet, but they can track auxiliary information for the purposes of verification. We write $\pi.f$ to denote the value of field f of π and $\pi[f{:=}n]$ for the packet obtained from π by updating field f to hold n. We let Pk denote the (finite) set of all packets.

Syntax. ProbNetKAT terms can be divided into two classes: *predicates* ($t, u, \ldots$) and *programs* ($p, q, \ldots$). Primitive predicates include *tests* ($f{=}n$) and the Boolean constants *false* (drop) and *true* (skip). Compound predicates are formed using the usual Boolean connectives: disjunction ($t \mathbin{\&} u$), conjunction ($t \mathbin{;} u$), and negation ($\neg t$). Primitive programs include *predicates* (t) and *assignments* ($f{\leftarrow}n$). The original version of the language also provides a dup primitive, which logs the current state of the packet, but the history-free fragment omits this operation. Compound programs can be formed using *parallel composition* ($p \mathbin{\&} q$), *sequential composition* ($p \mathbin{;} q$), and *iteration* (p^*). In addition, the *probabilistic choice* operator $p \oplus_r q$ executes p with probability r and q with probability $1 - r$, where r is rational, $0 \le r \le 1$. We sometimes use an n-ary version and omit the r's: $p_1 \oplus \cdots \oplus p_n$ executes a p_i chosen uniformly at random. In addition to these core constructs (summarized in Figure 2), many other useful constructs can be derived. For example, mutable local variables (*e.g.*, $\mathtt{up}_i$, used to track link health in §2), can be desugared into the language:

$$\textbf{var}\ f{\leftarrow}n\ \textbf{in}\ p \ \triangleq\ f{\leftarrow}n \mathbin{;} p \mathbin{;} f{\leftarrow}0$$

Here f is a field that is local to p. The final assignment $f{\leftarrow}0$ sets the value of f to a canonical value, "erasing" it after the field goes out of scope. We often use local variables to record extra information for verification—*e.g.*, recording whether a packet traversed a given switch allows reasoning about simple waypointing and isolation properties, even though the history-free fragment of ProbNetKAT does not model paths directly.

Guarded fragment. Conditionals and while loops can be encoded using union and iteration:

$$\textbf{if}\ t\ \textbf{then}\ p\ \textbf{else}\ q \ \triangleq\ t \mathbin{;} p \mathbin{\&} \neg t \mathbin{;} q$$
$$\textbf{while}\ t\ \textbf{do}\ p \ \triangleq\ (t \mathbin{;} p)^* \mathbin{;} \neg t$$

Note that these constructs use the predicate t as a *guard*, resolving the inherent nondeterminism in the union and iteration operators. Our implementation handles programs in the guarded fragment of the language—i.e., with loops and conditionals but without union and iteration—though we will develop the theory in full generality here, to make connections to previous work on ProbNetKAT clearer. We believe this restriction is acceptable from a practical perspective, as the main purpose of union and iteration is to encode forwarding tables and network-wide processing, and the guarded variants can often perform the same task. A notable exception is multicast, which cannot be expressed in the guarded fragment.

Naturals	$n ::= 0 \mid 1 \mid 2 \mid \cdots$	
Fields	$f ::= f_1 \mid \cdots \mid f_k$	
Packets	$\mathsf{Pk} \ni \pi ::= \{f_1 = n_1, \ldots, f_k = n_k\}$	
Probabilities	$r \in [0,1] \cap \mathbb{Q}$	
Predicates	$t, u ::= \mathsf{drop}$	*False*
	$\mid \mathsf{skip}$	*True*
	$\mid f{=}n$	*Test*
	$\mid t \mathbin{\&} u$	*Disjunction*
	$\mid t \mathbin{;} u$	*Conjunction*
	$\mid \neg t$	*Negation*
Programs	$p, q ::= t$	*Filter*
	$\mid f{\leftarrow}n$	*Assignment*
	$\mid p \mathbin{\&} q$	*Union*
	$\mid p \mathbin{;} q$	*Sequence*
	$\mid p \oplus_r q$	*Choice*
	$\mid p^*$	*Iteration*

Figure 2. ProbNetKAT Syntax.

Semantics. Previous work on ProbNetKAT [11] modeled history-free programs as maps $2^{\mathsf{Pk}} \to \mathcal{D}(2^{\mathsf{Pk}})$, where $\mathcal{D}(2^{\mathsf{Pk}})$ denotes the set of probability distributions on 2^{Pk}. This semantics is useful for establishing fundamental properties of the language, but we will need a more explicit representation to build a practical verification tool. Since the set of packets is finite, probability distributions over sets of packets are discrete and can be characterized by a *probability mass function*, $f : 2^{\mathsf{Pk}} \to [0,1]$ such that $\sum_{b \subseteq \mathsf{Pk}} f(b) = 1$. It will be convenient to view f as a *stochastic vector* of non-negative entries that sum to 1.

A program, which maps inputs a to distributions over outputs, can then be represented by a square matrix indexed by Pk in which the stochastic vector corresponding to input a appears as the a-th row. Thus, we can interpret a program p as a matrix $\mathcal{B}[\![p]\!] \in [0,1]^{2^{\mathsf{Pk}} \times 2^{\mathsf{Pk}}}$ indexed by packet sets, where the matrix entry $\mathcal{B}[\![p]\!]_{ab}$ gives the probability that p produces output $b \in 2^{\mathsf{Pk}}$ on input $a \in 2^{\mathsf{Pk}}$. The rows of the matrix $\mathcal{B}[\![p]\!]$ are stochastic vectors, each encoding the distribution produced for an input set a; such a matrix is called *right-stochastic*, or simply stochastic. We write $\mathbb{S}(2^{\mathsf{Pk}})$ for the set of right-stochastic matrices indexed by 2^{Pk}.

$$\boxed{\mathcal{B}[\![p]\!] \in \mathbb{S}(2^{\mathsf{Pk}})}$$

$$\begin{aligned}
\mathcal{B}[\![\mathsf{drop}]\!]_{ab} &\triangleq [b = \varnothing] \\
\mathcal{B}[\![\mathsf{skip}]\!]_{ab} &\triangleq [a = b] \\
\mathcal{B}[\![f{=}n]\!]_{ab} &\triangleq [b = \{\pi \in a \mid \pi.f = n\}] \\
\mathcal{B}[\![\neg t]\!]_{ab} &\triangleq [b \subseteq a] \cdot \mathcal{B}[\![t]\!]_{a,a-b} \\
\mathcal{B}[\![f{\leftarrow}n]\!]_{ab} &\triangleq [b = \{\pi[f := n] \mid \pi \in a\}] \\
\mathcal{B}[\![p \mathbin{\&} q]\!]_{ab} &\triangleq \sum_{c,d} [c \cup d = b] \cdot \mathcal{B}[\![p]\!]_{a,c} \cdot \mathcal{B}[\![q]\!]_{a,d} \\
\mathcal{B}[\![p \mathbin{;} q]\!] &\triangleq \mathcal{B}[\![p]\!] \cdot \mathcal{B}[\![q]\!] \\
\mathcal{B}[\![p \oplus_r q]\!] &\triangleq r \cdot \mathcal{B}[\![p]\!] + (1-r) \cdot \mathcal{B}[\![q]\!] \\
\mathcal{B}[\![p^*]\!]_{ab} &\triangleq \lim_{n\to\infty} \mathcal{B}[\![p^{(n)}]\!]_{ab}
\end{aligned}$$

Figure 3. ProbNetKAT Semantics. The notation $\mathcal{B}[\![p]\!]_{ab}$ denotes the probability that p produces b on input a.

Figure 3 defines an interpretation of ProbNetKAT programs as stochastic matrices; the Iverson bracket $[\varphi]$ is 1 if φ is true, and 0 otherwise. Deterministic program primitives are interpreted as $\{0,1\}$-matrices—*e.g.*, the program primitive drop is interpreted as the following stochastic matrix:

$$\mathcal{B}[\![\mathsf{drop}]\!] = \begin{array}{c} \\ \varnothing \\ \vdots \\ a_n \end{array} \begin{array}{c} \begin{array}{cccc} \varnothing & b_2 & \dots & b_n \end{array} \\ \begin{bmatrix} 1 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 1 & 0 & \cdots & 0 \end{bmatrix} \end{array} \qquad \begin{tikzcd} \boxed{a_2} \arrow[dr, "1"] & \\ \vdots & \boxed{a_1 = \varnothing} \arrow[loop right, "1"] \\ \boxed{a_n} \arrow[ur, "1"'] & \end{tikzcd} \tag{1}$$

which assigns all probability mass to the $\varnothing$-column. Similarly, skip is interpreted as the identity matrix. Sequential composition can be interpreted as matrix product,

$$\mathcal{B}[\![p \mathbin{;} q]\!]_{ab} = \sum_c \mathcal{B}[\![p]\!]_{ac} \cdot \mathcal{B}[\![q]\!]_{cb} = (\mathcal{B}[\![p]\!] \cdot \mathcal{B}[\![q]\!])_{ab}$$

which reflects the intuitive semantics of composition: to step from a to b in $\mathcal{B}[\![p \mathbin{;} q]\!]$, one must step from a to an intermediate state c in $\mathcal{B}[\![p]\!]$, and then from c to b in $\mathcal{B}[\![q]\!]$.

As the picture in (1) suggests, a stochastic matrix $B \in \mathbb{S}(2^{\mathsf{Pk}})$ can be viewed as a *Markov chain* (MC)—i.e., a probabilistic transition system with state space 2^{Pk}. The B_{ab} entry gives the probability that the system transitions from a to b.

Soundness. The matrix $\mathcal{B}[\![p]\!]$ is equivalent to the denotational semantics $[\![p]\!]$ defined in previous work [11].

Theorem 3.1 (Soundness)**.** *Let $a, b \in 2^{\mathsf{Pk}}$. The matrix $\mathcal{B}[\![p]\!]$ satisfies $\mathcal{B}[\![p]\!]_{ab} = [\![p]\!](a)(\{b\})$.*

Hence, checking program equivalence for p and q reduces to checking equality of the matrices $\mathcal{B}[\![p]\!]$ and $\mathcal{B}[\![q]\!]$.

Corollary 3.2. $[\![p]\!] = [\![q]\!]$ *if and only if* $\mathcal{B}[\![p]\!] = \mathcal{B}[\![q]\!]$.

In particular, because the Markov chains are all finite state, the transition matrices are finite dimensional with rational entries. Accordingly, program equivalence and other quantitative properties can be automatically verified provided we can compute the matrices for given programs. This is relatively straightforward for program constructs besides $\mathcal{B}[\![p^*]\!]$, whose matrix is defined in terms of a limit. The next section presents a closed-form definition of the stochastic matrix for this operator.

$$\begin{tikzcd}[column sep=small]
\langle p^*, a, b\rangle \arrow[r, dashed, "1"] \arrow[drr, "\mathcal{B}[\![p]\!]_{a,a'}"] & \langle \mathsf{skip} \mathbin{\&} p \mathbin{;} p^*, a, b\rangle \arrow[r, dashed, "1"] & \langle p \mathbin{;} p^*, a, b \cup a\rangle \arrow[d, dashed, "\mathcal{B}[\![p]\!]_{a,a'}"] \\
& & \langle p^*, a', b \cup a\rangle
\end{tikzcd}$$

Figure 4. The small-step semantics is given by a Markov chain with states $\langle$*program, input set, output accumulator*$\rangle$. The three dashed arrows can be collapsed into the single solid arrow, rendering the program component superfluous.

4 Computing Stochastic Matrices

The semantics developed in the previous section can be viewed as a "big-step" semantics in which a single step models the execution of a program from input to output. To compute the semantics of p^*, we will introduce a finer, "small-step" chain in which a transition models one iteration of the loop.

To build intuition, consider simulating p^* using a transition system with states given by triples $\langle p, a, b\rangle$ in which p is the program being executed, a is the set of (input) packets, and b is an accumulator that collects the output packets generated so far. To model the execution of p^* on input a, we start from the initial state $\langle p^*, a, \varnothing\rangle$ and unroll p^* one iteration according to the characteristic equation $p^* \equiv \mathsf{skip} \mathbin{\&} p \mathbin{;} p^*$, yielding the following transition:

$$\langle p^*, a, \varnothing\rangle \xrightarrow{\quad 1 \quad} \langle \mathsf{skip} \mathbin{\&} p \mathbin{;} p^*, a, \varnothing\rangle$$

Next, we execute both skip and $p \mathbin{;} p^*$ on the input set and take the union of their results. Executing skip yields the input set as output, with probability 1:

$$\langle \mathsf{skip} \mathbin{\&} p \mathbin{;} p^*, a, \varnothing\rangle \xrightarrow{\quad 1 \quad} \langle p \mathbin{;} p^*, a, a\rangle$$

Executing $p \mathbin{;} p^*$, executes p and feeds its output into p^*:

$$\forall a' : \quad \langle p \mathbin{;} p^*, a, a\rangle \xrightarrow{\mathcal{B}[\![p]\!]_{a,a'}} \langle p^*, a', a\rangle$$

At this point we are back to executing p^*, albeit with a different input set a' and some accumulated output packets. The resulting Markov chain is shown in Figure 4.

Note that as the first two steps of the chain are deterministic, we can simplify the transition system by collapsing all three steps into one, as illustrated in Figure 4. The program component can then be dropped, as it now remains constant across transitions. Hence, we work with a Markov chain over the state space $2^{\mathsf{Pk}} \times 2^{\mathsf{Pk}}$, defined formally as follows:

$$\begin{aligned} \mathcal{S}\llbracket p \rrbracket &\in \mathbb{S}(2^{\mathsf{Pk}} \times 2^{\mathsf{Pk}}) \\ \mathcal{S}\llbracket p \rrbracket_{(a,b),(a',b')} &\triangleq [b' = b \cup a] \cdot \mathcal{B}\llbracket p \rrbracket_{a,a'}. \end{aligned}$$

We can verify that the matrix $\mathcal{S}\llbracket p \rrbracket$ defines a Markov chain.

Lemma 4.1. *$\mathcal{S}\llbracket p \rrbracket$ is stochastic.*

Next, we show that each step in $\mathcal{S}\llbracket p \rrbracket$ models an iteration of p^*. Formally, the $(n+1)$-step of $\mathcal{S}\llbracket p \rrbracket$ is equivalent to the big-step behavior of the n-th unrolling of p^*.

Proposition 4.2. $\mathcal{B}\llbracket p^{(n)} \rrbracket_{a,b} = \sum_{a'} \mathcal{S}\llbracket p \rrbracket^{n+1}_{(a,\varnothing),(a',b)}$

Direct induction on the number of steps $n \geq 0$ fails because the hypothesis is too weak. We generalize from start states with empty accumulator to arbitrary start states.

Lemma 4.3. *Let p be program. Then for all $n \in \mathbb{N}$ and $a, b, b' \subseteq \mathsf{Pk}$, we have*

$$\sum_{a'} [b' = a' \cup b] \cdot \mathcal{B}\llbracket p^{(n)} \rrbracket_{a,a'} = \sum_{a'} \mathcal{S}\llbracket p \rrbracket^{n+1}_{(a,b),(a',b')}.$$

Proposition 4.2 then follows from Lemma 4.3 with $b = \varnothing$.

Intuitively, the long-run behavior of $\mathcal{S}\llbracket p \rrbracket$ approaches the big-step behavior of p^*: letting (a_n, b_n) denote the random state of the Markov chain $\mathcal{S}\llbracket p \rrbracket$ after taking n steps starting from $(a, \varnothing)$, the distribution of b_n for $n \to \infty$ is precisely the distribution of outputs generated by p^* on input a (by Proposition 4.2 and the definition of $\mathcal{B}\llbracket p^* \rrbracket$).

Closed form. The limiting behavior of finite state Markov chains has been well studied in the literature (*e.g.*, see Kemeny and Snell [20]). For so-called *absorbing* Markov chains, the limit distribution can be computed exactly. A state s of a Markov chain T is *absorbing* if it transitions to itself with probability 1,

$$\begin{tikzcd} \boxed{s} \arrow[loop right, "1"] \end{tikzcd} \qquad \text{(formally: } T_{s,s'} = [s = s'])$$

and a Markov chain $T \in \mathbb{S}(S)$ is *absorbing* if each state can reach an absorbing state:

$$\forall s \in S.\ \exists s' \in S, n \geq 0.\ T^n_{s,s'} > 0 \text{ and } T_{s',s'} = 1$$

The non-absorbing states of an absorbing MC are called *transient*. Assume T is absorbing with n_t transient states and n_a absorbing states. After reordering the states so that absorbing states appear first, T has the form

$$T = \begin{bmatrix} I & 0 \\ R & Q \end{bmatrix}$$

where I is the $n_a \times n_a$ identity matrix, R is an $n_t \times n_a$ matrix giving the probabilities of transient states transitioning to absorbing states, and Q is an $n_t \times n_t$ matrix specifying the probabilities of transitions between transient states. Since absorbing states never transition to transient states by definition, the upper right corner contains a $n_a \times n_t$ zero matrix.

From any start state, a finite state absorbing MC always ends up in an absorbing state eventually, *i.e.* the limit $T^\infty \triangleq \lim_{n\to\infty} T^n$ exists and has the form

$$T^\infty = \begin{bmatrix} I & 0 \\ A & 0 \end{bmatrix}$$

where the $n_t \times n_a$ matrix A contains the so-called *absorption probabilities*. This matrix satisfies the following equation:

$$A = (I + Q + Q^2 + \ldots)\, R$$

Intuitively, to transition from a transient state to an absorbing state, the MC can take an arbitrary number of steps between transient states before taking a single—and final—step into an absorbing state. The infinite sum $X \triangleq \sum_{n \geq 0} Q^n$ satisfies $X = I + QX$, and solving for X yields

$$X = (I - Q)^{-1} \quad \text{and} \quad A = (I - Q)^{-1} R. \tag{2}$$

(We refer the reader to Kemeny and Snell [20] for the proof that the inverse exists.)

Before we apply this theory to the small-step semantics $\mathcal{S}\llbracket - \rrbracket$, it will be useful to introduce some MC-specific notation. Let T be an MC. We write $s \xrightarrow{T}_n s'$ if s can reach s' in precisely n steps, *i.e.* if $T^n_{s,s'} > 0$; and we write $s \xrightarrow{T} s'$ if s can reach s' in some number of steps, *i.e.* if $T^n_{s,s'} > 0$ for some $n \geq 0$. Two states are said to *communicate*, denoted $s \overset{T}{\longleftrightarrow} s'$, if $s \xrightarrow{T} s'$ and $s' \xrightarrow{T} s$. The relation $\overset{T}{\longleftrightarrow}$ is an equivalence relation, and its equivalence classes are called *communication classes*. A communication class is *absorbing* if it cannot reach any states outside the class. Let $\Pr[s \xrightarrow{T}_n s']$ denote the probability $T^n_{s,s'}$. For the rest of the section, we fix a program p and abbreviate $\mathcal{B}\llbracket p \rrbracket$ as B and $\mathcal{S}\llbracket p \rrbracket$ as S. We also define *saturated states*, those where the accumulator has stabilized.

Definition 4.4. A state (a, b) of S is called *saturated* if b has reached its final value, *i.e.* if $(a, b) \xrightarrow{S} (a', b')$ implies $b' = b$.

After reaching a saturated state, the output of p^* is fully determined. The probability of ending up in a saturated state with accumulator b, starting from an initial state $(a, \varnothing)$, is

$$\lim_{n\to\infty} \sum_{a'} S^n_{(a,\varnothing),(a',b)}$$

and, indeed, this is the probability that p^* outputs b on input a by Proposition 4.2. Unfortunately, we cannot directly compute this limit since saturated states are not necessarily absorbing. To see this, consider $p^* = (f{\leftarrow}0 \oplus_{1/2} f{\leftarrow}1)^*$ over a single $\{0, 1\}$-valued field f. Then S has the form

$$\begin{tikzcd} & 0,0 \arrow[loop right] \arrow[dd] & 0,\{0,1\} \arrow[loop right] \arrow[dd, leftrightarrow] \\ 0,\varnothing \arrow[ur] \arrow[dr] & & \\ & 1,0 \arrow[r] \arrow[uur] & 1,\{0,1\} \arrow[loop right] \end{tikzcd}$$

where all edges are implicitly labeled with $\frac{1}{2}$, and 0 and 1 denote the packets with f set to 0 and 1 respectively. We omit states not reachable from $(0, \varnothing)$. The right-most states are saturated, but they communicate and are thus not absorbing.

To align saturated and absorbing states, we can perform a quotient of this Markov chain by collapsing the communicating states. We define an auxiliary matrix,

$$U_{(a,b),(a',b')} \triangleq [b' = b] \cdot \begin{cases} [a' = \varnothing] & \text{if } (a, b) \text{ is saturated} \\ [a' = a] & \text{else} \end{cases}$$

which sends a saturated state (a, b) to a canonical saturated state $(\varnothing, b)$ and acts as the identity on all other states. In our example, the modified chain SU is as follows:

$$\begin{tikzcd}
 & 0,0 \arrow[loop right] \arrow[d] & 0,\{0,1\} \arrow[dr] & \\
0,\varnothing \arrow[ur] \arrow[dr] & & & \varnothing,\{0,1\} \arrow[loop right] \\
 & 1,0 \arrow[uur] \arrow[r] & 1,\{0,1\} \arrow[ur] &
\end{tikzcd}$$

and indeed is absorbing, as desired.

Lemma 4.5. *S, U, and SU are* monotone *in the sense that: $(a, b) \xrightarrow{S} (a', b')$ implies $b \subseteq b'$ (and similarly for U and SU).*

Proof. By definition (S and U) and by composition (SU). □

Next, we show that SU is an absorbing MC:

Proposition 4.6. *Let $n \geq 1$.*

1. *$(SU)^n = S^n U$*
2. *SU is an absorbing MC with absorbing states $\{(\varnothing, b)\}$.*

Arranging the states (a, b) in lexicographically ascending order according to $\subseteq$ and letting $n = |2^{\mathsf{Pk}}|$, it then follows from Proposition 4.6.2 that SU has the form

$$SU = \begin{bmatrix} I_n & 0 \\ R & Q \end{bmatrix}$$

where, for $a \neq \varnothing$, we have

$$(SU)_{(a,b),(a',b')} = \begin{bmatrix} R & Q \end{bmatrix}_{(a,b),(a',b')}.$$

Moreover, SU converges and its limit is given by

$$(SU)^{\infty} \triangleq \begin{bmatrix} I_n & 0 \\ (I - Q)^{-1} R & 0 \end{bmatrix} = \lim_{n\to\infty} (SU)^n. \quad (3)$$

Putting together the pieces, we can use the modified Markov chain SU to compute the limit of S.

Theorem 4.7 (Closed Form)**.** *Let $a, b, b' \subseteq \mathsf{Pk}$. Then*

$$\lim_{n\to\infty} \sum_{a'} S^n_{(a,b),(a',b')} = (SU)^{\infty}_{(a,b),(\varnothing,b')}.$$

The limit exists and can be computed exactly, in closed-form.

5 Implementation

We have implemented McNetKAT as an embedded DSL in OCaml in roughly 10KLoC. The frontend provides functions for defining and manipulating ProbNetKAT programs and for generating such programs automatically from network topologies encoded using Graphviz. These programs can then be analyzed by one of two backends: the *native backend* (PNK), which compiles programs to (symbolically represented) stochastic matrices; or the *PRISM-based backend* (PPNK), which emits inputs for the state-of-the-art probabilistic model checker PRISM [24].

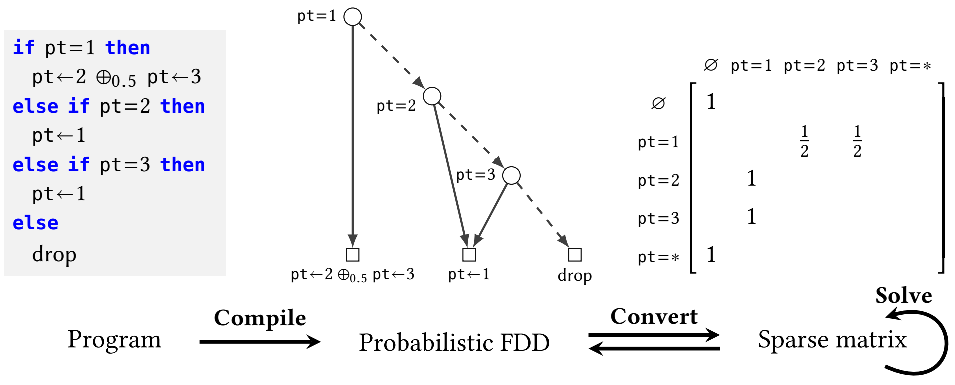
Pragmatic restrictions. Although our semantics developed in §3 and §4 theoretically supports computations on sets of packets, a direct implementation would be prohibitively expensive—the matrices are indexed by the powerset 2^{Pk} of the universe of all possible packets! To obtain a practical analysis tool, we restrict the state space to single packets. At the level of syntax, we restrict to the guarded fragment of ProbNetKAT, *i.e.* to programs with conditionals and while loops, but without union and iteration. This ensures that no proper packet sets are ever generated, thus allowing us to work over an exponentially smaller state space. While this restriction does rule out some uses of ProbNetKAT—most notably, modeling multicast—we did not find this to be a serious limitation because multicast is relatively uncommon in probabilistic networking. If needed, multicast can often be modeled using multiple unicast programs.

5.1 Native Backend

The native backend compiles a program to a symbolic representation of its big step matrix. The translation, illustrated in Figure 5, proceeds as follows. First, we translate atomic programs to Forwarding Decision Diagrams (FDDs), a symbolic data structure based on Binary Decision Diagrams (BDDs) that encodes sparse matrices compactly [34]. Second, we translate composite programs by first translating each subprogram to an FDD and then merging the results using standard BDD algorithms. Loops require special treatment: we (i) convert the FDD for the body of the loop to a sparse stochastic matrix, (ii) compute the semantics of the loop by using an optimized sparse linear solver [7] to solve the system from §4, and finally (iii) convert the resulting matrix back to an FDD. We use exact rational arithmetic in the frontend and FDD-backend to preempt concerns about numerical precision, but trust the linear algebra solver UMFPACK (based on 64 bit floats) to provide accurate solutions.[1] Our implementation relies on several optimizations; we detail two of the more interesting ones below.

Probabilistic FDDs. Binary Decision Diagrams [1] and variants thereof [13] have long been used in verification and

[1]UMFPACK is a mature library powering widely-used scientific computing packages such as MATLAB and SciPy.



```
if pt=1 then
  pt←2 ⊕0.5 pt←3
else if pt=2 then
  pt←1
else if pt=3 then
  pt←1
else
  drop
```

$$\begin{array}{c|ccccc} & \varnothing & \texttt{pt=1} & \texttt{pt=2} & \texttt{pt=3} & \texttt{pt=*} \\ \hline \varnothing & 1 & & & & \\ \texttt{pt=1} & & & \frac{1}{2} & \frac{1}{2} & \\ \texttt{pt=2} & & 1 & & & \\ \texttt{pt=3} & & 1 & & & \\ \texttt{pt=*} & 1 & & & & \end{array}$$

Figure 5. Implementation using FDDs and a sparse linear algebra solver.

model checking to represent large state spaces compactly. A variant called Forwarding Decision Diagrams (FDDs) [34] was previously developed specifically for the networking domain, but only supported deterministic behavior. In this work, we extended FDDs to probabilistic FDDs. A probabilistic FDD is a rooted directed acyclic graph that can be understood as a control-flow graph. Interior nodes test packet fields and have outgoing true- and false- branches, which we visualize by solid lines and dashed lines in Figure 5. Leaf nodes contain distributions over *actions*, where an action is either a set of modifications or a special action drop. To interpret an FDD, we start at the root node with an initial packet and traverse the graph as dictated by the tests until a leaf node is reached. Then, we apply each action in the leaf node to the packet. Thus, an FDD represents a function of type $\mathsf{Pk} \to \mathcal{D}(\mathsf{Pk} + \varnothing)$, or equivalently, a stochastic matrix over the state space $\mathsf{Pk} + \varnothing$ where the $\varnothing$-row puts all mass on $\varnothing$ by convention. Like BDDs, FDDs respect a total order on tests and contain no isomorphic subgraphs or redundant tests, which enables representing sparse matrices compactly.

Dynamic domain reduction. As Figure 5 shows, we do not have to represent the state space $\mathsf{Pk} + \varnothing$ explicitly even when converting into sparse matrix form. In the example, the state space is represented by *symbolic packets* $\texttt{pt} = 1$, $\texttt{pt} = 2$, $\texttt{pt} = 3$, and $\texttt{pt} = *$, each representing an *equivalence class* of packets. For example, $\texttt{pt} = 1$ can represent all packets π satisfying $\pi.\texttt{pt} = 1$, because the program treats all such packets in the same way. The packet $\texttt{pt} = *$ represents the set $\{\pi \mid \pi.\texttt{pt} \notin \{1, 2, 3\}\}$. The symbol $*$ can be thought of as a wildcard that ranges over all values not explicitly represented by other symbolic packets. The symbolic packets are chosen dynamically when converting an FDD to a matrix by traversing the FDD and determining the set of values appearing in each field, either in a test or a modification. Since FDDs never contain redundant tests or modifications, these sets are typically of manageable size.

5.2 PRISM backend

PRISM is a mature probabilistic model checker that has been actively developed and improved for the last two decades. The tool takes as input a Markov chain model specified symbolically in PRISM's input language and a property specified using a logic such as Probabilistic CTL, and outputs the probability that the model satisfies the property. PRISM supports various types of models including finite state Markov chains, and can thus be used as a backend for reasoning about ProbNetKAT programs using our results from §3 and §4. Accordingly, we implemented a second backend that translates ProbNetKAT to PRISM programs. While the native backend computes the big step semantics of a program—a costly operation that may involve solving linear systems to compute fixed points—the PRISM backend is a purely syntactic transformation; the heavy lifting is done by PRISM itself.

A PRISM program consists of a set of bounded variables together with a set of transition rules of the form

$$\phi \ \longrightarrow \ p_1 \cdot u_1 + \cdots + p_k \cdot u_k$$

where ϕ is a Boolean predicate over the variables, the p_i are probabilities that must sum up to one, and the u_i are sequences of variable updates. The predicates are required to be mutually exclusive and exhaustive. Such a program encodes a Markov chain whose state space is given by the finite set of variable assignments and whose transitions are dictated by the rules: if ϕ is satisfied under the current assignment σ and σ_i is obtained from σ by performing update u_i, then the probability of a transition from σ to σ_i is p_i.

It is easy to see that any PRISM program can be expressed in ProbNetKAT, but the reverse direction is slightly tricky: it requires the introduction of an additional variable akin to a program counter to emulate ProbNetKAT's control flow primitives such as loops and sequences. As an additional challenge, we must be economical in our allocation of the program counter, since the performance of model checking is very sensitive to the size of the state space.

We address this challenge in three steps. First, we translate the ProbNetKAT program to a finite state machine using a

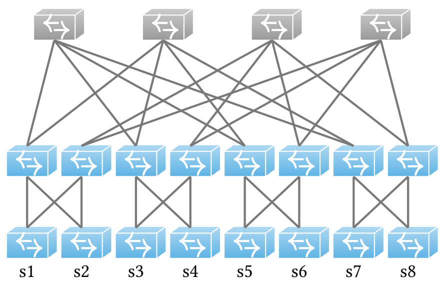


Figure 6. A FatTree topology with $p = 4$.

Thompson-style construction [37]. Each edge is labeled with a predicate ϕ, a probability p_i, and an update u_i, subject to the following well-formedness conditions:

1. For each state, the predicates on its outgoing edges form a partition.
2. For each state and predicate, the probabilities of all outgoing edges guarded by that predicate sum to one.

Intuitively, the state machine encodes the control-flow graph.

This intuition serves as the inspiration for the next translation step, which collapses each basic block of the graph into a single state. This step is crucial for reducing the state space, since the state space of the initial automaton is linear in the size of the program. Finally, we obtain a PRISM program from the automaton as follows: for each state s with adjacent predicate ϕ and ϕ-guarded outgoing edges $s \xrightarrow{\phi/p_i/u_i} t_i$ for $1 \leq i \leq k$, produce a PRISM rule

$$(pc{=}s \wedge \phi) \;\longrightarrow\; p_1 \cdot (u_1 \,; pc{\leftarrow}t_1) + \cdots + p_k \cdot (u_k \,; pc{\leftarrow}t_k).$$

The well-formedness conditions of the state machine guarantee that the resulting program is a valid PRISM program. With some care, the entire translation can be implemented in linear time. Indeed, McNetKAT translates all programs in our evaluation to PRISM in under a second.

6 Evaluation

To evaluate McNetKAT we conducted experiments on several benchmarks including a family of real-world data center topologies and a synthetic benchmark drawn from the literature [15]. We evaluated McNetKAT's scalability, characterized the effect of optimizations, and compared performance against other state-of-the-art tools. All McNetKAT running times we report refer to the time needed to compile programs to FDDs; the cost of comparing FDDs for equivalence and ordering, or of computing statistics of the encoded distributions, is negligible. All experiments were performed on machines with 16-core, 2.6 GHz Intel Xeon E5-2650 processors with 64 GB of memory.

Scalability on FatTree topologies. We first measured the scalability of McNetKAT by using it to compute network models for a series of FatTree topologies of increasing size. FatTrees [2] (see also Figure 6) are multi-level, multi-rooted

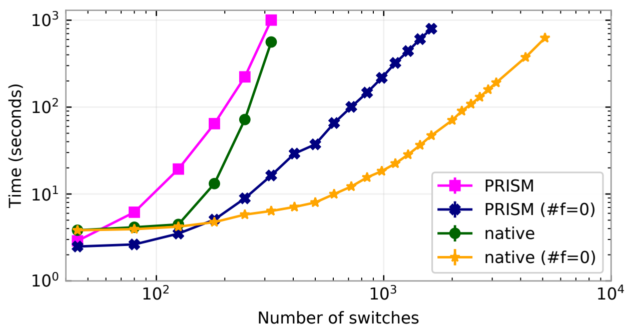


Figure 7. Scalability on a family of data center topologies.

trees that are widely used as topologies in modern data centers. FatTrees can be specified in terms of a parameter p corresponding to the number of ports on each switch. A p-ary FatTree connects $\frac{1}{4}p^3$ servers using $\frac{5}{4}p^2$ switches. To route packets, we used a form of Equal-Cost Multipath Routing (ECMP) that randomly maps traffic flows onto shortest paths. We measured the time needed to construct the stochastic matrix representation of the program on a single machine using two backends (native and PRISM) and under two failure models (no failures and independent failures with probability 1/1000).

Figure 7 depicts the results, several of which are worth discussing. First, the native backend scales quite well: in the absence of failures ($f = 0$), it scales to a network with 5000 switches in approximately 10 minutes. This result shows that McNetKAT is able to handle networks of realistic size. Second, the native backend consistently outperforms the PRISM backend. We conjecture that the native backend is able to exploit algebraic properties of the ProbNetKAT program to better parallelize the job. Third, performance degrades in the presence of failures. This is to be expected—failures lead to more complex probability distributions which are nontrivial to represent and manipulate.

Parallel speedup. One of the contributors to McNetKAT's good performance is its ability to parallelize the computation of stochastic matrices across multiple cores in a machine, or even across machines in a cluster. Intuitively, because a network is a large collection of mostly independent devices, it is possible to model its global behavior by first modeling the behavior of each device in isolation, and then combining the results to obtain a network-wide model. In addition to speeding up the computation, this approach can also reduce memory usage, often a bottleneck on large inputs.

To facilitate parallelization, we added an n-ary disjoint branching construct to ProbNetKAT:

```
case sw=1 then $p_1$ else
case sw=2 then $p_2$ else
  . . .
case sw=$n$ then $p_n$
```

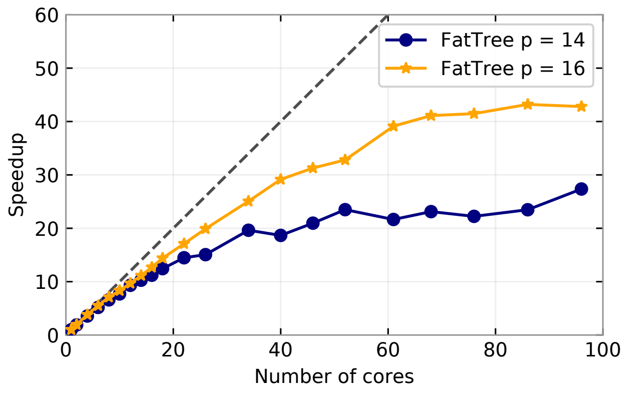


Figure 8. Speedup due to parallelization.

Semantically, this construct is equivalent to a cascade of conditionals; but the native backend compiles it in parallel using a map-reduce-style strategy, using one process per core by default.

To evaluate the impact of parallelization, we compiled two representative FatTree models ($p = 14$ and $p = 16$) using ECMP routing on an increasing number of cores. With m cores, we used one master machine together with $r = \lceil m/16 - 1 \rceil$ remote machines, adding machines one by one as needed to obtain more physical cores. The results are shown in Figure 8. We see near linear speedup on a single machine, cutting execution time by more than an order of magnitude on our 16-core test machine. Beyond a single machine, the speedup depends on the complexity of the submodels for each switch—the longer it takes to generate the matrix for each switch, the higher the speedup. For example, with a $p = 16$ FatTree, we obtained a 30x speedup using 40 cores across 3 machines.

Comparison with other tools. Bayonet [15] is a state-of-the-art tool for analyzing probabilistic networks. Whereas McNetKAT has a native backend tailored to the networking domain and a backend based on a probabilistic model checker, Bayonet programs are translated to a general-purpose probabilistic language which is then analyzed by the symbolic inference engine PSI [16]. Bayonet's approach is more general, as it can model queues, state, and multi-packet interactions under an asynchronous scheduling model. It also supports Bayesian inference and parameter synthesis. Moreover, Bayonet is fully symbolic whereas McNetKAT uses a numerical linear algebra solver [7] (based on floating point arithmetic) to compute limits.

To evaluate how the performance of these approaches compares, we reproduced an experiment from the Bayonet paper that analyzes the reliability of a simple routing scheme in a family of "chain" topologies indexed by k, as shown in Figure 9.

For $k = 1$, the network consists of four switches organized into a diamond, with a single link that fails with probability

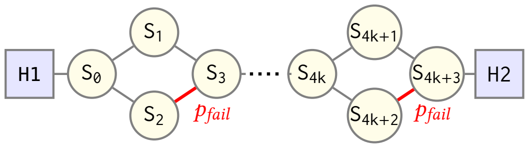


Figure 9. Chain topology

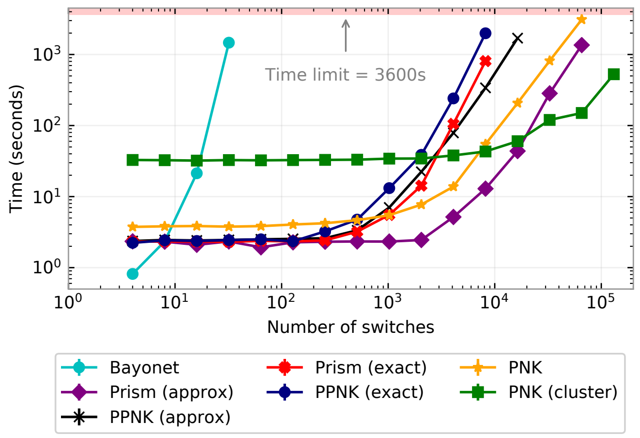


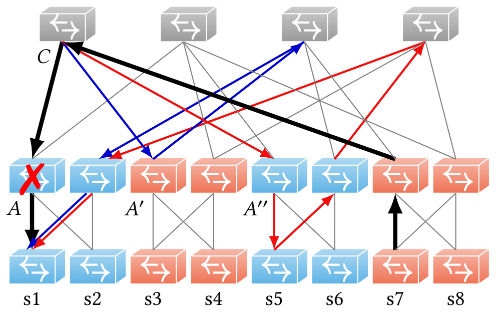
Figure 10. Scalability on chain topology.

$p_{fail} = 1/1000$. For $k > 1$, the network consists of k diamonds linked together into a chain as shown in Figure 9. Within each diamond, switch S_0 forwards packets with equal probability to switches S_1 and S_2, which in turn forward to switch S_3. However, S_2 drops the packet if the link to S_3 fails. We analyze the probability that a packet originating at `H1` is successfully delivered to `H2`. Our implementation does not exploit the regularity of these topologies.

Figure 10 gives the running time for several tools on this benchmark: Bayonet, hand-written PRISM, ProbNetKAT with the PRISM backend (PPNK), and ProbNetKAT with the native backend (PNK). Further, we ran the PRISM tools in exact and approximate mode, and we ran the ProbNetKAT backend on a single machine and on the cluster. Note that both axes in the plot are log-scaled.

We see that Bayonet scales to 32 switches in about 25 minutes, before hitting the one hour time limit and 64 GB memory limit at 48 switches. ProbNetKAT answers the same query for 2048 switches in under 10 seconds and scales to over 65000 switches in about 50 minutes on a single core, or just 2.5 minutes using a cluster of 24 machines. PRISM scales similarly to ProbNetKAT, and performs best using the hand-written model in approximate mode.

Overall, this experiment shows that for basic network verification tasks, ProbNetKAT's domain-specific backend based on specialized data structures and an optimized linear-algebra library [7] can outperform an approach based on a general-purpose solver.



k	$\widehat{M}(F10_0, f_k) \equiv$ *teleport*	$\widehat{M}(F10_3, f_k) \equiv$ *teleport*	$\widehat{M}(F10_{3,5}, f_k) \equiv$ *teleport*
0	✓	✓	✓
1	✗	✓	✓
2	✗	✓	✓
3	✗	✗	✓
4	✗	✗	✗
∞	✗	✗	✗

k	`compare` F10$_0$ F10$_3$	`compare` F10$_3$ F10$_{3,5}$	`compare` F10$_{3,5}$ *teleport*
0	≡	≡	≡
1	<	≡	≡
2	<	≡	≡
3	<	<	≡
4	<	<	<
∞	<	<	<

Figure 11. (a) AB FatTree topology with $p = 4$. (b) Evaluating k-resilience. (c) Comparing schemes under k failures.

7 Case Study: Data Center Fault-Tolerance

In this section, we go beyond benchmarks and present a case study that illustrates the utility of McNetKAT for probabilistic reasoning. Specifically, we model the F10 [26] data center design in ProbNetKAT and verify its key properties.

Data center resilience. An influential measurement study by Gill et al. [17] showed that data centers experience frequent failures, which have a major impact on application performance. To address this challenge, a number of data center designs have been proposed that aim to simultaneously achieve high throughput, low latency, and fault tolerance.

F10 topology. F10 uses a novel topology called an *AB FatTree*, see Figure 11(a), that enhances a traditional FatTree [2] with additional backup paths that can be used when failures occur. To illustrate, consider routing from s_7 to s_1 in Figure 11(a) along one of the shortest paths (in thick black). After reaching the core switch C in a standard FatTree (recall Figure 6), if the aggregation switch on the downward path failed, we would need to take a 5-hop detour (shown in red) that goes down to a different edge switch, up to a different core switch, and finally down to s_1. In contrast, an AB FatTree [26] modifies the wiring of the aggregation later to provide shorter detours—e.g., a 3-hop detour (shown in blue) for the previous scenario.

F10 routing. F10's routing scheme uses three strategies to re-route packets after a failure occurs. If a link on the current path fails and an equal-cost path exists, the switch simply re-routes along that path. This approach is also known as *equal-cost multi-path routing* (ECMP). If no shortest path exist, it uses a 3-hop detour if one is available, and otherwise falls back to a 5-hop detour if necessary.

We implemented this routing scheme in ProbNetKAT in several steps. The first, F10$_0$, approximates the hashing behavior of ECMP by randomly selecting a port along one of the shortest paths to the destination. The second, F10$_3$, improves the resilience of F10$_0$ by augmenting it with 3-hop re-routing—e.g., consider the blue path in Figure 11(a). We find a port on C that connects to a different aggregation switch A' and forward the packet to A'. If there are multiple such ports which have not failed, we choose one uniformly at random. The third, F10$_{3,5}$, attempts 5-hop re-routing in cases where F10$_3$ is unable to find a port on C whose adjacent link is up—e.g., consider the red path in Figure 11(a). The 5-hop rerouting strategy requires a flag to distinguish packets taking a detour from regular packets.

F10 network and failure model. We model the network as discussed in §2, focusing on packets destined to switch 1:

$$M(p) \triangleq in ; \textbf{do } (p ; t) \textbf{ while } (\neg \texttt{sw}{=}1)$$

McNetKAT automatically generates the topology program t from a Graphviz description. The ingress predicate *in* is a disjunction of switch-port tests over all ingress locations. Adding the failure model and some setup code to declare local variables tracking the health of individual links yields the complete network model:

$$\widehat{M}(p, f) \triangleq \textbf{var } \texttt{up}_1 {\leftarrow} 1 \textbf{ in } \ldots \textbf{ var } \texttt{up}_d {\leftarrow} 1 \textbf{ in } M(f ; p)$$

Here, d is the maximum degree of a topology node. The entire model measures about 750 lines of ProbNetKAT code.

To evaluate the effect of different kinds of failures, we define a family of failure models f_k indexed by the maximum number of failures $k \in \mathbb{N} \cup \{\infty\}$ that may occur, where links fail otherwise independently with probability pr; we leave pr implicit. To simplify the analysis, we focus on failures occurring on downward paths (note that F10$_0$ is able to route around failures on the upward path, unless the topology becomes disconnected).

Verifying refinement. Having implemented F10 as a series of three refinements, we would expect the probability of packet delivery to increase in each refinement, but not to achieve perfect delivery in an unbounded failure model f_∞. Formally, we should have

$$\begin{aligned} \text{drop} < \widehat{M}(F10_0, f_\infty) &< \widehat{M}(F10_3, f_\infty) \\ &< \widehat{M}(F10_{3,5}, f_\infty) < \textit{teleport} \end{aligned}$$

where *teleport* moves the packet directly to its destination, and $p < q$ means the probability assigned to every input-output pair by q is greater than the probability assigned by p. We confirmed that these inequalities hold using McNetKAT.

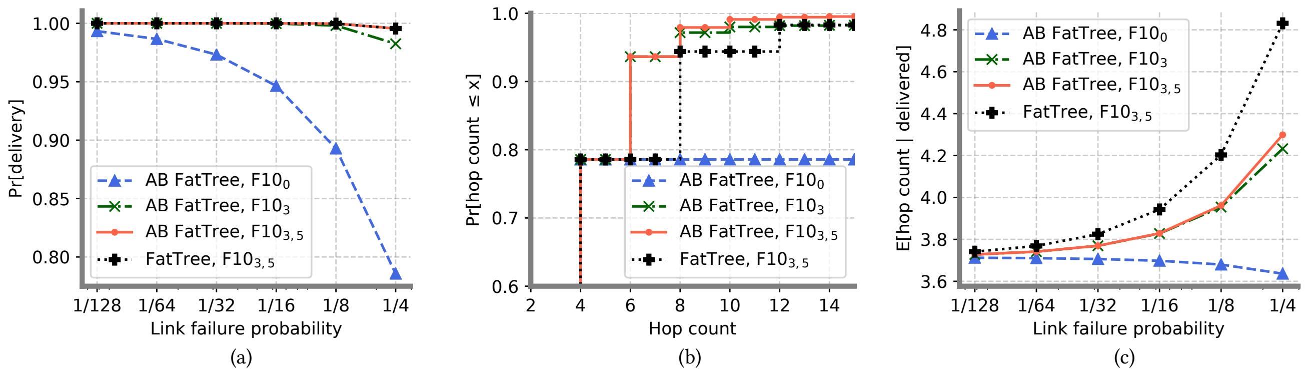


Figure 12. Case study results ($k = \infty$): (a) Probability of delivery vs. link-failure probability; (b) Increased path length due to resilience ($pr = 1/4$); (c) Expected hop-count conditioned on delivery.

Verifying k-resilience. Resilience is the key property satisfied by F10. By using McNetKAT, we were able to automatically verify that F10 is resilient to up to three failures in the AB FatTree Figure 11(a). To establish this property, we increased the parameter k in our failure model f_k while checking equivalence with teleportation (i.e., perfect delivery), as shown in Figure 11(b). The simplest scheme $F10_0$ drops packets when a failure occurs on the downward path, so it is 0-resilient. The $F10_3$ scheme routes around failures when a suitable aggregation switch is available, hence it is 2-resilient. Finally, the $F10_{3,5}$ scheme routes around failures as long as any aggregation switch is reachable, hence it is 3-resilient. If the schemes are not equivalent to *teleport*, we can still compare the relative resilience of the schemes using the refinement order, as shown in Figure 11(c). Our implementation also enables precise, quantitative comparisons. For example, Figure 12(a) considers a failure model in which an unbounded number of failures can occur. We find that $F10_0$'s delivery probability dips significantly as the failure probability increases, while both $F10_3$ and $F10_{3,5}$ continue to ensure high delivery probability by routing around failures.

Analyzing path stretch. Routing schemes based on detours achieve a higher degree of resilience at the cost of increasing the lengths of forwarding paths. We can quantify this increase by augmenting our model with a counter that is incremented at each hop and analyzing the expected path length. Figure 12(b) shows the cumulative distribution function of latency as the fraction of traffic delivered within a given hop count. On AB FatTree, $F10_0$ delivers ≈80% of the traffic in 4 hops, since the maximum length of a shortest path from any edge switch to $s1$ is 4 and $F10_0$ does not attempt to recover from failures. $F10_3$ and $F10_{3,5}$ deliver the same amount of traffic when limited to at most 4 hops, but they can deliver significantly more traffic using 2 additional hops by using 3-hop and 5-hop paths to route around failures. $F10_3$ also delivers more traffic with 8 hops—these are the cases when $F10_3$ performs 3-hop re-routing twice for a single packet as it encountered failure twice. We can also show that on a standard FatTree, $F10_{3,5}$ failures have a higher impact on latency. Intuitively, the topology does not support 3-hop re-routing. This finding supports a key claim of F10: the topology and routing scheme should be co-designed to avoid excessive path stretch. Finally, Figure 12(c) shows the expected path length conditioned on delivery. As the failure probability increases, the probability of delivery for packets routed via the core layer decreases for $F10_0$. Thus, the distribution of delivered packets shifts towards 2-hop paths via an aggregation switch, so the expected hop-count decreases.

8 Related Work

The most closely related system to McNetKAT is Bayonet [15]. In contrast to the domain-specific approach followed in this paper, Bayonet uses a general-purpose probabilistic programming language and inference tool [16]. Such an approach, which reuses existing techniques, is naturally appealing. In addition, Bayonet is more expressive than McNetKAT: it supports asynchronous scheduling, stateful transformations, and probabilistic inference, making it possible to model richer phenomena, such as congestion due to packet-level interactions in queues. Of course, the extra generality does not come for free. Bayonet requires programmers to supply an upper bound on loops as the implementation is not guaranteed to find a fixed point. As discussed in §5, McNetKAT scales better than Bayonet on simple benchmarks. Another issue is that writing a realistic scheduler appears challenging, and one might also need to model host-level congestion control protocols to obtain accurate results. Currently Bayonet programs use deterministic or uniform schedulers and model only a few packets at a time [14].

Prior work on ProbNetKAT [35] gave a measure-theoretic semantics and an implementation that approximated programs using sequences of monotonically improving estimates. While these estimates were proven to converge in the limit, [35] offered no guarantees about the convergence rate. In fact, there are examples where the approximations do not converge after any finite number of steps, which is obviously undesirable in a tool. The implementation only scaled to 10s of switches. In contrast, this paper presents a straightforward and implementable semantics; the implementation computes limits precisely in closed form, and it scales to real-world networks with thousands of switches. McNetKAT achieves this by restricting to the guarded and history-free fragment of ProbNetKAT, sacrificing the ability to reason about multicast and path-properties directly. In practice this sacrifice seems well worth the payoff: multicast is somewhat uncommon, and we can often reason about path-properties by maintaining extra state in the packets. In particular, McNetKAT can still model the examples studied in previous work by Smolka et al. [35].

Our work is the latest in a long line of techniques using Markov chains as a tool for representing and analyzing probabilistic programs. For an early example, see the seminal paper of Sharir et al. [32]. Markov chains are also used in many probabilistic model checkers, such as PRISM [23].

Beyond networking applications, there are connections to other work on verification of probabilistic programs. Di Pierro, Hankin, and Wiklicky used probabilistic abstract interpretation to statically analyze probabilistic λ-calculus [8]; their work was extended to a language *pWhile*, using a store and program location state space similar to Sharir et al. [32]. However, they do not deal with infinite limiting behavior beyond stepwise iteration, and do not guarantee convergence. Olejnik, Wicklicky, and Cheraghchi provided a probabilistic compiler *pwc* for a variation of *pWhile* [29]; their optimizations could potentially be useful for McNetKAT. A recent survey by Gordon et al. [18] shows how to give semantics for probabilistic processes using stationary distributions of Markov chains, and studies convergence. Similar to our approach, they use absorbing strongly connected components to represent termination. Finally, probabilistic abstract interpretation is also an active area of research [39]; it would be interesting to explore applications to ProbNetKAT.

9 Conclusion

This paper presents a scalable tool for verifying probabilistic networks based on a new semantics for the history-free fragment of ProbNetKAT in terms of Markov chains. Natural directions for future work include further optimization of our implementation—e.g., using Bayesian networks to represent joint distributions compactly. We are also interested in applying McNetKAT to other systems that implement algorithms for randomized routing [22, 33], load balancing [10], traffic monitoring [31], anonymity [9], and network neutrality [41], among others.

Acknowledgments

We are grateful to the anonymous reviewers and our shepherd Michael Greenberg for their feedback and help in improving the paper. Thanks also to Jonathan DiLorenzo for suggesting improvements to the paper and for helping us locate a subtle performance bug, and to the Bellairs Research Institute of McGill University for providing a wonderful research environment. This work was supported in part by the National Science Foundation under grants NeTS-1413972 and AiTF-1637532, by the European Research Council under grant 679127, by a Facebook TAV award, by a Royal Society Wolfson fellowship, and a gift from Keysight.

References

[1] S. B. Akers. 1978. Binary Decision Diagrams. *IEEE Trans. Comput.* 27, 6 (June 1978), 509–516. https://doi.org/10.1109/TC.1978.1675141

[2] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. 2008. A Scalable, Commodity Data Center Network Architecture. In *ACM SIGCOMM Computer Communication Review*, Vol. 38. ACM, 63–74.

[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 *POPL*. 113–126.

[4] Manav Bhatia, Mach Chen, Sami Boutros, Marc Binderberger, and Jeffrey Haas. 2014. Bidirectional Forwarding Detection (BFD) on Link Aggregation Group (LAG) Interfaces. RFC 7130. https://doi.org/10.17487/RFC7130

[5] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. 2014. P4: Programming Protocol-Independent Packet Processors. *SIGCOMM CCR* 44, 3 (July 2014), 87–95.

[6] Martin Casado, Nate Foster, and Arjun Guha. 2014. Abstractions for Software-Defined Networks. *CACM* 57, 10 (Oct. 2014), 86–95.

[7] Timothy A. Davis. 2004. Algorithm 832: UMFPACK V4.3—an Unsymmetric-pattern Multifrontal Method. *ACM Trans. Math. Softw.* 30, 2 (June 2004), 196–199. https://doi.org/10.1145/992200.992206

[8] Alessandra Di Pierro, Chris Hankin, and Herbert Wiklicky. 2005. Probabilistic λ-calculus and quantitative program analysis. *Journal of Logic and Computation* 15, 2 (2005), 159–179. https://doi.org/10.1093/logcom/exi008

[9] Roger Dingledine, Nick Mathewson, and Paul Syverson. 2004. Tor: The Second-generation Onion Router. In *USENIX Security Symposium (SSYM)*. 21–21.

[10] A. Dixit, P. Prakash, Y. C. Hu, and R. R. Kompella. 2013. On the impact of packet spraying in data center networks. In *IEEE INFOCOM*. 2130–2138.

[11] Nate Foster, Dexter Kozen, Konstantinos Mamouras, Mark Reitblatt, and Alexandra Silva. 2016. Probabilistic NetKAT. In *ESOP*. 282–309. https://doi.org/10.1007/978-3-662-49498-1_12

[12] Nate Foster, Dexter Kozen, Matthew Milano, Alexandra Silva, and Laure Thompson. 2015. A Coalgebraic Decision Procedure for NetKAT. In *POPL*. ACM, 343–355.

[13] M. Fujita, P. C. McGeer, and J. C.-Y. Yang. 1997. Multi-Terminal Binary Decision Diagrams: An Efficient DataStructure for Matrix Representation. *Form. Methods Syst. Des.* 10, 2-3 (April 1997), 149–169. https://doi.org/10.1023/A:1008647823331

[14] Timon Gehr, Sasa Misailovic, Petar Tsankov, Laurent Vanbever, Pascal Wiesmann, and Martin T. Vechev. 2018. Bayonet: Probabilistic Computer Network Analysis. Available at https://github.com/eth-sri/bayonet/.

[15] Timon Gehr, Sasa Misailovic, Petar Tsankov, Laurent Vanbever, Pascal Wiesmann, and Martin T. Vechev. 2018. Bayonet: probabilistic inference for networks. In *ACM SIGPLAN PLDI*. 586–602.

[16] Timon Gehr, Sasa Misailovic, and Martin T. Vechev. 2016. PSI: Exact Symbolic Inference for Probabilistic Programs. 62–83.

[17] Phillipa Gill, Navendu Jain, and Nachiappan Nagappan. 2011. Understanding Network Failures in Data Centers: Measurement, Analysis, and Implications. In *ACM SIGCOMM*. 350–361.

[18] Andrew D Gordon, Thomas A Henzinger, Aditya V Nori, and Sriram K Rajamani. 2014. Probabilistic programming. In *Proceedings of the on Future of Software Engineering*. ACM, 167–181. https://doi.org/10.1145/2593882.2593900

[19] Peyman Kazemian, George Varghese, and Nick McKeown. 2012. Header Space Analysis: Static Checking for Networks. In *USENIX NSDI 2012*. 113–126. https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/kazemian

[20] John G Kemeny and James Laurie Snell. 1960. *Finite markov chains*. Vol. 356. van Nostrand Princeton, NJ.

[21] Ahmed Khurshid, Wenxuan Zhou, Matthew Caesar, and Brighten Godfrey. 2012. Veriflow: Verifying Network-Wide Invariants in Real Time. In *ACM SIGCOMM*. 467–472.

[22] Praveen Kumar, Yang Yuan, Chris Yu, Nate Foster, Robert Kleinberg, Petr Lapukhov, Chiun Lin Lim, and Robert Soulé. 2018. Semi-Oblivious Traffic Engineering: The Road Not Taken. In *USENIX NSDI*.

[23] M. Kwiatkowska, G. Norman, and D. Parker. 2011. PRISM 4.0: Verification of Probabilistic Real-time Systems. In *Proc. 23rd International Conference on Computer Aided Verification (CAV'11) (LNCS)*, G. Gopalakrishnan and S. Qadeer (Eds.), Vol. 6806. Springer, 585–591. https://doi.org/10.1007/978-3-642-22110-1_47

[24] Marta Z. Kwiatkowska, Gethin Norman, and David Parker. 2011. PRISM 4.0: Verification of Probabilistic Real-Time Systems. In *CAV*. 585–591.

[25] Jed Liu, William Hallahan, Cole Schlesinger, Milad Sharif, Jeongkeun Lee, Robert Soulé, Han Wang, Calin Cascaval, Nick McKeown, and Nate Foster. 2018. p4v: Practical Verification for Programmable Data Planes. In *SIGCOMM*. 490–503.

[26] Vincent Liu, Daniel Halperin, Arvind Krishnamurthy, and Thomas E Anderson. 2013. F10: A Fault-Tolerant Engineered Network. In *USENIX NSDI*. 399–412.

[27] Haohui Mai, Ahmed Khurshid, Rachit Agarwal, Matthew Caesar, P. Brighten Godfrey, and Samuel Talmadge King. 2011. Debugging the Data Plane with Anteater. In *ACM SIGCOMM*. 290–301.

[28] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. 2008. OpenFlow: Enabling Innovation in Campus Networks. *SIGCOMM CCR* 38, 2 (2008), 69–74.

[29] Maciej Olejnik, Herbert Wiklicky, and Mahdi Cheraghchi. 2016. Probabilistic Programming and Discrete Time Markov Chains. http://www.imperial.ac.uk/media/imperial-college/faculty-of-engineering/computing/public/MaciejOlejnik.pdf

[30] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C. Snoeren. 2015. Inside the Social Network's (Datacenter) Network. In *ACM SIGCOMM*. 123–137.

[31] Vyas Sekar, Michael K. Reiter, Walter Willinger, Hui Zhang, Ramana Rao Kompella, and David G. Andersen. 2008. CSAMP: A System for Network-wide Flow Monitoring. In *USENIX NSDI*. 233–246.

[32] Micha Sharir, Amir Pnueli, and Sergiu Hart. 1984. Verification of probabilistic programs. *SIAM J. Comput.* 13, 2 (1984), 292–314. https://doi.org/10.1137/0213021

[33] Rachee Singh, Manya Ghobadi, Klaus-Tycho Foerster, Mark Filer, and Phillipa Gill. 2018. RADWAN: Rate Adaptive Wide Area Network. In *ACM SIGCOMM*.

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

[35] Steffen Smolka, Praveen Kumar, Nate Foster, Dexter Kozen, and Alexandra Silva. 2017. Cantor Meets Scott: Semantic Foundations for Probabilistic Networks. In *POPL 2017*. https://doi.org/10.1145/3009837.3009843

[36] Steffen Smolka, Praveen Kumar, David M Kahn, Nate Foster, Justin Hsu, Dexter Kozen, and Alexandra Silva. 2019. Scalable Verification of Probabilistic Networks (Extended Version). *arXiv* (2019). arXiv:1904.08096

[37] Ken Thompson. 1968. Regular Expression Search Algorithm. *Commun. ACM* 11, 6 (1968), 419–422. https://doi.org/10.1145/363347.363387

[38] L. Valiant. 1982. A Scheme for Fast Parallel Communication. *SIAM J. Comput.* 11, 2 (1982), 350–361.

[39] Di Wang, Jan Hoffmann, and Thomas Reps. 2018. PMAF: An Algebraic Framework for Static Analysis of Probabilistic Programs. In *POPL 2018*. https://www.cs.cmu.edu/~janh/papers/WangHR17.pdf

[40] Geoffrey G. Xie, Jibin Zhan, David A. Maltz, Hui Zhang, Albert G. Greenberg, Gísli Hjálmtýsson, and Jennifer Rexford. 2005. On static reachability analysis of IP networks. In *INFOCOM*.

[41] Zhiyong Zhang, Ovidiu Mara, and Katerina Argyraki. 2014. Network Neutrality Inference. In *ACM SIGCOMM*. 63–74.