Bayonet: Probabilistic Inference for Networks

Timon Gehr
ETH Zurich
timon.gehr@inf.ethz.ch

Sasa Misailovic
UIUC
misailo@illinois.edu

Petar Tsankov
ETH Zurich
ptsankov@inf.ethz.ch

Laurent Vanbever
ETH Zurich
lvanbever@ethz.ch

Pascal Wiesmann
ETH Zurich
wipascal@student.ethz.ch

Martin Vechev
ETH Zurich
martin.vechev@inf.ethz.ch

Abstract

Network operators often need to ensure that important probabilistic properties are met, such as that the probability of network congestion is below a certain threshold. Ensuring such properties is challenging and requires both a suitable language for probabilistic networks and an automated procedure for answering probabilistic inference queries.

We present Bayonet, a novel approach that consists of: (i) a probabilistic network programming language and (ii) a system that performs probabilistic inference on Bayonet programs. The key insight behind Bayonet is to phrase the problem of probabilistic network reasoning as inference in existing probabilistic languages. As a result, Bayonet directly leverages existing probabilistic inference systems and offers a flexible and expressive interface to operators.

We present a detailed evaluation of Bayonet on common network scenarios, such as network congestion, reliability of packet delivery, and others. Our results indicate that Bayonet can express such practical scenarios and answer queries for realistic topology sizes (with up to 30 nodes).

CCS Concepts • **Mathematics of computing** → *Probabilistic reasoning algorithms*; • **Networks** → *Network simulations*;

Keywords Probabilistic Programming, Computer Networks

ACM Reference Format:
Timon Gehr, Sasa Misailovic, Petar Tsankov, Laurent Vanbever, Pascal Wiesmann, and Martin Vechev. 2018. Bayonet: Probabilistic Inference for Networks. In *Proceedings of 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'18)*. ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3192366.3192400

PLDI'18, June 18–22, 2018, Philadelphia, PA, USA
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5698-5/18/06…$15.00
https://doi.org/10.1145/3192366.3192400

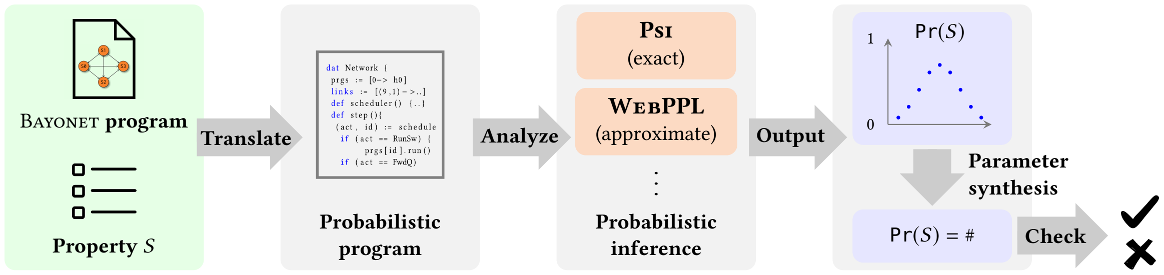
1 Introduction

Computer networks often exhibit probabilistic behaviors. Packets can get dropped because of random failures (devices or links) or congestion events. The paths that packets take are not necessarily deterministic, as network operators often employ probabilistic load-balancing techniques (e.g., ECMP [29]) to reduce network load. In addition, network traffic itself is also probabilistic – each host individually decides when, how much, and where to send packets.

There is little work in the literature on automatically reasoning about the effects of probabilistic behaviors in networks, an exception being the work of [61] (though as discussed in our related work, the two approaches differ substantially). While network simulators or emulators [11, 41, 56, 64] can be used to evaluate properties under uncertainty, their power is limited and they cannot provide statistical guarantees. Finally, many inherently deterministic network programming languages [3, 46, 53], routing protocols [49, 58], and verification tools [37, 38] require a developer to remove randomness from their assumptions, which can make the analyzed (deterministic) network models significantly different from the physical networks, rendering the analysis results inaccurate.

Key Challenges As a result, key challenges for effective and operator-friendly analysis of networks are: (i) the design of a language that allows operators to capture practical probabilistic network scenarios and properties, and (ii) an inference system capable of automated probabilistic reasoning about these scenarios. Addressing these challenges would allow operators to more quickly experiment with various adversarial scenarios and design more robust networks.

This Work We introduce Bayonet, a novel approach that consists of: (i) a probabilistic network programming language able to capture interesting and practical network scenarios, and (ii) a system that performs automated probabilistic inference on Bayonet programs. An important insight of Bayonet is that it phrases the problem of probabilistic network reasoning as inference in *standard* probabilistic programming systems, accomplished by translating Bayonet programs to probabilistic programs. This is an important insight because it allows Bayonet to directly take advantage of state-of-the-art probabilistic inference systems and



```
dat Network {
 prgs := [0-> h0]
 links := [(9,1)->..]
 def scheduler() {..}
 def step(){
  (act, id) := schedule
   if (act == RunSw) {
         prgs[id].run()
   if (act == FwdQ)
```

Figure 1. Probabilistic Inference for Networks with Bayonet.

algorithms. For instance, we show how to use Bayonet to automatically synthesize the configuration of a probabilistic network, such as link costs.

Flow of Probabilistic Inference and Use Cases Figure 1 presents the flow of probabilistic inference for networks using Bayonet. A Bayonet program captures the network topology, the network programs for each router, and a probabilistic property S. Bayonet translates these inputs to an underlying probabilistic program and computes the probability Pr(S) that the property S holds using existing probabilistic inference engines. Our design has the advantage that (i) it abstracts away the properties (e.g., queues) a network operator would normally need to somehow express in the unfamiliar lower-level probabilistic language, (ii) it decouples network specification from any individual probabilistic language and inference system, and (iii) it allows analysis of network integrity properties which would be harder to achieve in general-purpose languages.

We demonstrate the use of Bayonet for two kinds of scenarios: (i) *probabilistic analysis*: Bayonet can either output a concrete value for Pr(S) or check if this value is within the required bounds, shown in the bottom part of the last step in Figure 1, and (ii) *probabilistic synthesis*: Bayonet can automatically configure the probabilistic network by first obtaining a symbolic expression for Pr(S) parameterized by the unknown symbolic configuration parameters and then infer concrete values for these parameters that satisfy the property S with a developer-specified probability. In both scenarios, we leverage the insight that probabilistic programming systems can act as powerful solvers for problems in systems and networks akin to how SMT solvers have been successfully used in program verification.

Evaluation We show that Bayonet is effective for many common networking scenarios, such as computing the probability of congestion, probability of correct load-balancing, reliability of packets, and convergence in gossip protocols. We evaluate Bayonet on different choices of inputs, schedulers and network topologies (up to 30 nodes). Our results indicate that Bayonet's approach works for networks of realistic sizes: a recent analysis of 141 production networks indeed reported that 70% of them have 30 point-of-presences or less, each of which acting as one node [39].

Benefits over general PPL Bayonet provides three benefits over general probabilistic programming languages (PPL): (1) it checks various domain-specific integrity constraints (e.g., link queue capacities) that are harder to check in a general PPL, (2) it requires less code to express the desired functionality, and (3) it can be more easily compiled to general PPL, thus benefiting from different solvers.

Main Contributions Our main contributions are:

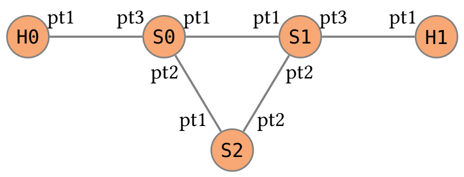
- The Bayonet language for specifying probabilistic interactions in networks (Sections 3.1 and 3.2).
- A query language for expressing probabilistic network properties (Section 3.3).
- The Bayonet system, which leverages state-of-the-art probabilistic inference engines by translating Bayonet programs and properties to these languages (Section 4).
- An experimental evaluation, which demonstrates the effectiveness of Bayonet on a set of 13 interesting network scenarios (Section 5).

2 Overview

In this section, we demonstrate Bayonet on a simple, but illustrative example, showing how to express a non-trivial network in the language. We use this example to demonstrate how Bayonet computes probability of *congestion*, which occurs whenever a packet is dropped because it exceeds the capacity of a network switch. Reasoning about congestion is challenging when routers forward traffic probabilistically (e.g., for load-balancing reasons) as each packet can be forwarded along one of many possible paths. We will also show how to automatically infer network configuration parameters so as to lower this probability, effectively automating a common traffic engineering task.

2.1 Example Probabilistic Network

We present the topology of our example network, along with the Bayonet programs for all network nodes (hosts and switches), in Figure 2. In this example, routers rely on the Open Shortest Path First (OSPF) protocol to compute



```
topology {
  nodes { H0, H1, S0, S1, S2 }
  links { (H0,pt1) <-> (S0,pt3),
    (S0,pt1) <-> (S1,pt1), (S0,pt2) <-> (S2,pt1),
    (S1,pt2) <-> (S2,pt2), (S1,pt3) <-> (H1,pt1) }
}
```

```
1  packet_fields { dst }
2  programs { H0 -> h0, H1 -> h1,
3             S0 -> s0, S1 -> s1,
4             S2 -> s2 }
5
6  def h0(pkt, pt) state pkt_cnt(0) {
7    if pkt_cnt < 3 {
8      new;
9      pkt.dst = H1;
10     fwd(1);
11     pkt_cnt = pkt_cnt + 1;
12    } else { drop; }
13 }
14 def h1(pkt, pt) state pkt_cnt(0) {
15   pkt_cnt = pkt_cnt + 1;
16   drop;
17 }
18 def s2(pkt, pt) {
19   if pt == 1 {
20     fwd(2);
21   } else {
22     fwd(1);
23   }
24 }
```

```
24 def s0(pkt, pt)
25   state route1(0), route2(0) {
26   if pt == 1 {
27     fwd(3);
28   } else if pt == 2 {
29     if pkt.dst == H0 {
30       fwd(3);
31     } else {
32       fwd(1);
33     }
34   } else if pt == 3 {
35     route1 = COST_01 ;
36     route2 = COST_02 + COST_21 ;
37     if route1 < route2 or
38       (route1 == route2
39         and flip(1/2)) {
40       fwd(1);
41     } else {
42       fwd(2);
43     }
44   }
45 }
```

```
46 def s1(pkt, pt)
47   state route1(0), route2(0) {
48   if pt == 1 {
49     fwd(3);
50   } else if pt == 2 {
51     if pkt.dst == H1 {
52       fwd(3);
53     } else {
54       fwd(1);
55     }
56   } else if pt == 3 {
57     route1 = COST_01;
58     route2 = COST_02 + COST_21;
59     if route1 < route2 or
60        (route1 == route2
61         and flip(1/2)) {
62       fwd(1);
63     } else {
64       fwd(2);
65     }
66   }
67 }
```

Figure 2. Topology of the network example and its Bayonet representation

point-to-point paths with minimum costs, and forward packets along these paths. If there are multiple paths with the same cost, routers load balance traffic among all of them uniformly, a widely-used load-balancing technique known as Equal-Cost MultiPath routing (ECMP). For simplicity, we assume the load-balancing decision is done for each packet individually; a per-flow decision is easy to model: since all packets in a flow have identical headers, one can simply base the decision on the hash of the packet headers.

Network Topology A network topology defines the *network nodes*, which can be hosts or switches, and the links that interconnect them. The network in Figure 2 has two hosts, `H0` and `H1`, and three switches, `S0`, `S1`, and `S2`. Each link between nodes is represented by two *ports*. For instance, `S0`'s port `pt1` and `S1`'s port `pt1` determine the link `S0`↔`S1`. Each node has an *input queue*, which contains packets received on the node's input ports, and an *output queue* with packets to be dispatched to the node's output ports. In our example, all queues have a capacity of 2 packets.

A packet can follow multiple paths though the network. For instance, a packet from `H0` to `H1` can follow the direct route `S0`⟶`S1` or the route `S0`⟶`S2`⟶`S1`. Switch `S0` compares the costs of the two routes to decide which one is followed.

Capturing Probabilistic Hosts and Switches To capture the behavior of a network node, the network operator specifies a Bayonet *program*, which defines how the node processes packets (e.g., program `s1` defines the behavior of the switch `S1`). Since the network nodes read *packet fields* while handling packets, the operator also specifies any relevant fields, such as `dst`, the destination IP address of the packet.

Figure 2 presents the Bayonet programs for the five network nodes. Each program takes two inputs: `pkt` is the packet on top of the input queue and `pt` is the (integer) port on which the packet was received. A program may use the keyword `state` to define additional variables whose values are preserved while processing packets; e.g., at line 6, program `h0` declares variable `pkt_cnt` and initializes it to `0`. Bayonet programs manipulate packets using: `new`, which creates a new packet on the node's input queue; `drop`, which removes the packet on top of the input queue; and `fwd(pt)`, which forwards the packet on top of the input queue to port `pt`.

A program can run on a node whenever there is at least one packet in its input queue. In the beginning, there is a single packet in the input queue of host `H0` and all other queues are empty.

We now describe the Bayonet programs associated to the hosts and switches in our example network. The Bayonet program associated with host `H0` is declared at line 6. This program creates a new packet, sets its destination to `H1`, and sends it to port `pt1` (which is connected to switch `S0`). The host performs this action at most three times, since it is guarded by the condition `pkt_cnt < 3` and `pkt_cnt` is increased every time a packet is sent.

The program of switch `S0` is declared at line 24. Switch `S0` forwards packets received on port `pt1` to host `H0` via port `pt3`. Packets received on port `pt2` and destined to `H0` are also forwarded to `H0` via port `pt3`, while the remaining packets received on `pt2` are forwarded to switch `S1` via port `pt1`. If switch `S0` receives a packet from `H0` on port `pt3`, it forwards this packet either directly over the link `S0`⟶`S1` or over link `S0`⟶`S2`. To decide which link to forward the packet to, the switch computes the costs associated with the two candidate routes based on the costs of the links `S0`⟶`S1`, `S0`⟶`S2`, and `S2`⟶`S1` (denoted as symbolic constants `COST_01`, `COST_02`, and `COST_21`). The program computes the costs of the two routes and stores the result in state variables `route1` and `route2`; see lines 35-36. If both routes have equal costs, it randomly selects a route via a call to `flip` (line 39). If the direct route to `S1` has smaller cost, or it is selected by `flip`, the packet is sent to `S1` (line 40). Otherwise, the packet is sent to `S2` (line 42).

The remaining Bayonet programs associated to the other nodes are fairly simple: the program of switch `S1` is analogous to that of `S0`; switch `S2` forwards packets received from `S0` to host `H1`; host `H1` counts and discards received packets.

Probabilistic Scheduling of Network Nodes The host and switches in a real network execute asynchronously. In Bayonet, this asynchrony is captured via a probabilistic scheduler, which selects an action (a statement in a node's program) with a specified probability. For flexibility, Bayonet allows operators to specify a probabilistic scheduler as a program written in the Bayonet language. We further discuss scheduling and specify a uniform scheduler in Section 3. In practice, the scheduler might be used to model properties of the equipment, such as link transmission delays and switch speed.

2.2 Analysis of Probabilistic Networks

We now show how to express probabilistic properties for networks, such as probability of congestion, and demonstrate how Bayonet automatically analyzes such properties using exact and approximate probabilistic inference engines. We discuss further properties and examples in our evaluation (Section 5).

Expressing Relevant Properties We first demonstrate how to express the probability of congestion for our network example in Figure 2. The probability of congestion is given by the probability that some packets are dropped due to overflowing the capacity of the queues. For our network example, this can be expressed with the query

```
probability(pkt_cnt@H1 < 3)
```

Here, the value of `pkt_cnt@H1` corresponds to the number of packets received by host `H1`, as specified in its program. Since host `H0` sends exactly 3 packets to host `H1`, if `H1` receives less than 3 packets then congestion has occurred. The Bayonet system automatically computes that for link costs `COST_01 = 2`, `COST_02 = 1`, and `COST_21 = 1`, we have

$$\texttt{probability(pkt\_cnt@H1 < 3)} = \frac{30378810105265}{67706637778944} \approx 0.45$$

To capture different properties, network operators can replace the predicate S in the query `probability(S)` with a predicate that specifies the property of interest.

Further, Bayonet allows operators to write queries of the form `expectation(V)`, which computes the *expected value* of the expression `V`. For instance, the query

```
expectation(pkt_cnt@H1)
```

captures the expected number of packets received by host `H1`. As we will show in our evaluation section, these two types of queries enable network operators to express a range of interesting network properties, including correctness of load-balancing, reliability of packet delivery, convergence of gossip protocols, and traffic distribution.

Answering Queries using Exact or Approximate Inference Given a query `probability(S)`, Bayonet computes the probability that predicate S is satisfied on the network's final state (i.e., the state from which no further steps can be executed). To reason about such queries, Bayonet translates the programs associated with all network nodes into a probabilistic program (in a standard probabilistic programming language), encodes the query, and runs existing probabilistic inference engines. While most probabilistic engines are approximate [25, 26, 43], several recent systems support exact probabilistic inference [24, 52]. Bayonet leverages the power of these systems as well as any future advances (instead of reinventing them from scratch).

Concretely, our Bayonet system translates node programs to programs in the PSI language [24]; we present this translation step in Section 4. The PSI framework has the option to either perform exact symbolic inference with its own solver or to use approximate inference by translating PSI programs to WebPPL [27], a popular framework that uses approximate, sampling-based methods. Thus, Bayonet can analyze probabilistic networks using either exact solvers (especially suitable for experimenting with smaller networks and when exact results are desirable) or solvers based on approximate

Probability of congestion	Symbolic constraint
$\frac{30378810105265}{67706637778944} \approx 0.4487$	`COST_01 = COST_02+COST_21`
$\frac{491806403}{1088391168} \approx 0.4519$	`COST_01 < COST_02+COST_21`
$\frac{2025575442161}{4231664861184} \approx 0.4787$	`COST_01 > COST_02+COST_21`

Figure 3. Probability of congestion as a function of the symbolic parameters `COST_01`, `COST_02`, and `COST_21`.

inference (if one would like to scale the analysis to larger networks where reduced precision may be acceptable).

Bayesian Inference with Observations Bayonet supports analysis of probabilistic networks that include additional *evidence* about the state of hosts and switches. This is useful as existing network management systems routinely sample network traffic processed by switches for billing and statistics [13, 57]. With Bayonet, operators can leverage these sampled packets as observations to determine whether a given property is met or not, e.g. whether the network correctly load-balances packets along multiple paths. Specifically, network operators can encode evidence using `observe` statements and use Bayonet to perform classic Bayesian reasoning to answer this question (Section 5.5).

2.3 Synthesis of Network Configuration Parameters

To improve network utilization, such as minimizing the probability of congestion, operators often have to manually tailor the routers' configurations. In our example, this means finding specific values for the symbolic constants `COST_01`, `COST_02`, and `COST_21`, which represent link costs.

To support this scenario, Bayonet allows network operators to leave relevant configuration parameters symbolic. By using the PSI exact solver, Bayonet can operate on symbolic constraints and output the probability of congestion (or other queries) as a function of such parameters. The resulting constraints can then be used to synthesize concrete values that optimize a given property of interest (e.g. congestion). For instance, the probability of congestion

$$p(\texttt{COST\_01}, \texttt{COST\_02}, \texttt{COST\_21})$$

for our example which is produced by Bayonet, is given in Figure 3. The minimum congestion probability, ≈ 0.4487, is obtained when the condition `COST_01 = COST_02 + COST_21` holds. Note that when this condition holds, then switch `S0` load-balances packets destined to `H1` along the two possible paths to reach `H1`. Concrete values that satisfy this condition can be obtained using existing solvers, such as Mathematica or Z3 [17].

3 The Bayonet Language

In this section, we present the Bayonet language. We first define the syntax and then describe its semantics.

3.1 Specifying Probabilistic Networks in Bayonet

We depict the syntax of the Bayonet language in Figure 4.

Network Topology We consider networks with multiple hosts and switches. We do not distinguish between hosts and switches and refer to them as (network) nodes. Each network node is uniquely identified with a natural number between 1 and k and has up to l ports. A node's port defines an interface, which can be connected to another node's interface. An interface is a pair $(n, \texttt{pt})$, where n is a node and `pt` is a port, and a link is a pair of two interfaces. The network topology is defined by all nodes and their links.

Packets and Queues Switches process packets based on their header fields. Bayonet allows operators to define the fields necessary to specify the behavior of hosts and switches, such as `id`, `src`, `dst`, and `protocol`.

To model the input and output queues of network nodes, we introduce packet queues (p, c), which are lists of packets p with a designated capacity c. We write $(\texttt{pkt}, \texttt{pt}){:}(p, c)$ to denote a queue whose head element is $(\texttt{pkt}, \texttt{pt})$, and denote the enqueue operation as:

$$(p, c){:}(\texttt{pkt}, \texttt{pt}) = \begin{cases} (p \mathbin{+\!\!+} [(\texttt{pkt}, \texttt{pt})], c) & \text{if length}(p) < c \\ (p, c) & \text{otherwise} \end{cases}$$

where $+\!\!+$ denotes list concatenation. Note that the enqueue operation does not modify the queue if the number of packets has reached the capacity of the queue.

Probabilistic Packet-processing Programs The behavior of hosts and switches is defined by *probabilistic packet processing programs*, which are programs that send and receive packets; we call these Bayonet programs. A Bayonet program takes as input a packet and outputs zero or more packets, to be forwarded to other nodes in the network. Bayonet programs are probabilistic programs that feature designated network commands, such as `drop` and `fwd`. To record information relevant to a node's behavior, each network node may keep local state in its local variables; e.g. a switch records which of its interfaces are up since it cannot forward packets along interfaces that are down.

Along with standard arithmetic expressions, Bayonet supports expressions that draw values from probability distributions; e.g., the expression `flip`(p) draws a value from a Bernoulli(p) distribution. Drawing such values is necessary to model probabilistic events, such as link and component failures, and probabilistic load-balancing protocols, such as ECMP, which are widely used by existing switches. Furthermore, random values can be drawn to specify *prior distributions* on uncertain properties of the network.

Network Topology

(Nodes)	n	$\in$	$Nodes = \{1, \ldots, k\}$, for $k \in \mathbb{N}$
(Ports)	$\texttt{pt}$	$\in$	$Ports = \{1, \ldots, l\}$, for $l \in \mathbb{N}$
(Interfaces)	$Ifaces$	$\subseteq$	$Nodes \times Ports$
(Links)	$links$	$\subseteq$	$\binom{Ifaces}{2}$, such that $\forall i \in Ifaces.\ \lvert\{l \in links \mid i \in l\}\rvert \leq 1$
(Topology)	G	$=$	$(Nodes, links)$

Packets and Packet Queues

(Fields)	f	$\in$	$Fields = \{\texttt{id}, \texttt{src}, \texttt{dst}, \texttt{protocol}, \ldots\}$
(Values)	v	$\in$	$Vals = \mathbb{Q}$
(Packets)	$\texttt{pkt}$	$\in$	$Pkts = \{\texttt{pkt} \mid \texttt{pkt} : Fields \to Vals\}$
(Queues)	Q	$\in$	$Queues = \{(p, c) \in (Pkts \times Ports)^* \times \mathbb{N} \mid \text{length}(p) \leq c\}$

Probabilistic Packet-processing Programs

(Vars)	x	$\in$	$Vars$
(Values)	v	$\in$	$Vals = \mathbb{Q}$
(State)	σ	$\in$	$Vars \to Vals$
(Dist)	$dist$	$::=$	$\texttt{flip} \mid \texttt{uniformInt} \mid \cdots$
(AExpr)	e	$::=$	$v \mid x \mid \texttt{pkt}.f \mid \texttt{pt} \mid e + e \mid v \cdot e \mid dist(e)$
(BExpr)	b	$::=$	$e == e \mid e < e \mid b \ \texttt{and}\ b \mid \texttt{not}\ b$
(Stmt)	s	$::=$	$\texttt{new} \mid \texttt{drop} \mid \texttt{dup} \mid \texttt{fwd}(\texttt{pt}) \mid x = e \mid \texttt{pkt}.f = e$ $\mid \texttt{assert}(b) \mid \texttt{observe}(b) \mid \texttt{skip} \mid s; s$ $\mid \texttt{if}\ b\,\{\,s\,\}\texttt{else}\{\,s\,\} \mid \texttt{while}\ b\,\{\,s\,\}$
(Config)	C	$::=$	$\langle \sigma, Q_{\texttt{IN}}, Q_{\texttt{OUT}}, s \rangle \in Configs$

Network

(Network)	N_{init}	$::=$	$(G, \Pi\colon Nodes \to Stmt, \sigma_{sched}, C_1, \ldots, C_k)$

Figure 4. Syntax of the Bayonet Language

The statement $\texttt{observe}(e)$ specifies a condition that has been observed to hold, while $\texttt{assert}(e)$ is used to assert that a given condition e holds. If an observation fails, this indicates that the current branch of the program has not been realized and should be disregarded. If an assertion fails, the program terminates in the special state $\perp$. Bayonet also supports standard sequence, conditional, and looping statements. We assume that programs terminate with probability 1.

A *configuration* of a network node is given by a tuple $\langle \sigma, Q_{\texttt{IN}}, Q_{\texttt{OUT}}, s \rangle$, where σ is the node's state that assigns values to variables, $Q_{\texttt{IN}}$ and $Q_{\texttt{OUT}}$ are the node's input and, respectively, output queues, and s is the sequence of unexecuted statements of the node's Bayonet program.

Creating a Network To specify a network and define the initial state of all hosts and switches, operators must specify the initial *network configuration* $N_{init} = (G, \Pi, \sigma_s, C_1, \ldots C_k)$, where $G = (Nodes, links)$ determines the network topology, $\Pi : Nodes \to Stmt$ assigns a Bayonet program to each network node, σ_s defines the scheduler state (see below), and $(C_1, \ldots, C_k)$ are the local node configurations. To avoid clutter, we omit the topology G and the assignment of Bayonet programs Π when writing network configurations.

3.2 Semantics of Probabilistic Networks

We now present the semantics of Bayonet networks.

Expressions The meaning of arithmetic expressions is as expected. An expression e is evaluated at a given state σ and at a pair $(\texttt{pkt}, \texttt{pt})$ of a packet and a port (which are at the head of the input queue $Q_{\texttt{IN}}$). For instance, evaluating a local variable x at state σ returns the value $\sigma(x)$, i.e. the value assigned to x according to the local state σ. The expression $\texttt{pkt}.f$ returns the values stored at field f of the packet $\texttt{pkt}$. All expressions except $dist(e)$ are deterministic. We write

$$\langle \sigma, (\texttt{pkt}, \texttt{pt}) \rangle, e \xrightarrow{p} \langle \sigma, (\texttt{pkt}, \texttt{pt}) \rangle, e'$$

to denote that e evaluates to e' with probability p.

Local Network Node Semantics The semantics of Bayonet programs is defined in terms of transition steps which follow the form $C \xrightarrow{p} C'$, where $C = \langle \sigma, Q_{\texttt{IN}}, Q_{\texttt{OUT}}, s \rangle$ and $C' = \langle \sigma', Q'_{\texttt{IN}}, Q'_{\texttt{OUT}}, s' \rangle$ are node configurations. Each step may change the state σ of the network node, its input $Q_{\texttt{IN}}$ and output $Q_{\texttt{OUT}}$ queues, and its sequence s of unexecuted statements. Further, each step $C \xrightarrow{p} C'$ is labeled with a probability p, which denotes that if the node starts at configuration C then it transitions to C' with probability p.

In Figure 5, we provide the small-step semantic rules. For example, Rule L-Drop defines the meaning of statement $\texttt{drop}$: it removes the packet on top of the input queue. This step is deterministic, and it is labeled with probability 1 accordingly. Rule L-Fwd states that statement $\texttt{fwd}(\texttt{pt}')$ moves packet $\texttt{pkt}$ from input queue $Q_{\texttt{IN}}$ to output queue $Q_{\texttt{OUT}}$. The input port $\texttt{pt}$ is changed to $\texttt{pt}'$. The statement $\texttt{new}$ enqueues a packet at the node's input queue, and $\texttt{dup}$ duplicates the packet on top of the queue. Note that statements such as $\texttt{drop}$ and $\texttt{fwd}$ can be applied only if the input queue $Q_{\texttt{IN}}$ is nonempty. Further, the statements $\texttt{new}$ and $\texttt{dup}$ cannot cause the queue to exceed its capacity because the enqueue operation leaves full queues intact. The remaining rules are fairly standard.

Global Network Semantics The global network semantics defines how the network nodes are executed and how packets move across the network. These global steps are scheduled by a *probabilistic scheduler*, which selects actions of the form $\lambda \in \{\text{Run}, \text{Fwd}\} \times \{1, \ldots, k\}$. An action (Run, i) means that the ith network node executes a number of steps of its Bayonet program, and an action (Fwd, i) means that a packet processed by the ith node is delivered to its destination (i.e., to some node's input queue).

A (global) network configuration is a tuple $(\sigma_s, C_1, .., C_k)$, where σ_s is the state of the probabilistic scheduler, and C_i are the (local) network node configurations. Given scheduler state σ_s, and local configurations $C_1, \ldots, C_k$, the probability with which the scheduler selects action λ and updates its

$$\frac{\texttt{pkt} \in \mathit{Pkts} \qquad \forall f \in \mathit{Fields}.\ \texttt{pkt}(f) = 0}{\langle \sigma, Q_{\texttt{IN}}, Q_{\texttt{OUT}}, \texttt{new} \rangle \xrightarrow{1} \langle \sigma, (\texttt{pkt}, 0) : Q_{\texttt{IN}}, Q_{\texttt{OUT}}, \texttt{skip} \rangle}\ \textsc{L-New}$$

$$\frac{}{\langle \sigma, (\texttt{pkt}, \texttt{pt}) : Q_{\texttt{IN}}, Q_{\texttt{OUT}}, \texttt{drop} \rangle \xrightarrow{1} \langle \sigma, Q_{\texttt{IN}}, Q_{\texttt{OUT}}, \texttt{skip} \rangle}\ \textsc{L-Drop}$$

$$\frac{}{\langle \sigma, (\texttt{pkt}, \texttt{pt}) : Q_{\texttt{IN}}, Q_{\texttt{OUT}}, \texttt{dup} \rangle \xrightarrow{1} \langle \sigma, (\texttt{pkt}, \texttt{pt}) : (\texttt{pkt}, \texttt{pt}) : Q_{\texttt{IN}}, Q_{\texttt{OUT}}, \texttt{skip} \rangle}\ \textsc{L-Dup}$$

$$\frac{}{\langle \sigma, (\texttt{pkt}, \texttt{pt}) : Q_{\texttt{IN}}, Q_{\texttt{OUT}}, \texttt{fwd}(\texttt{pt}') \rangle \xrightarrow{1} \langle \sigma, Q_{\texttt{IN}}, Q_{\texttt{OUT}} : (\texttt{pkt}, \texttt{pt}'), \texttt{skip} \rangle}\ \textsc{L-Fwd}$$

$$\frac{\langle \sigma, (\texttt{pkt}, \texttt{pt}) \rangle, e \xrightarrow{p} e'}{\langle \sigma, (\texttt{pkt}, \texttt{pt}) : Q_{\texttt{IN}}, Q_{\texttt{OUT}}, x = e \rangle \xrightarrow{p} \langle \sigma, (\texttt{pkt}, \texttt{pt}) : Q_{\texttt{IN}}, Q_{\texttt{OUT}}, x = e' \rangle}\ \textsc{L-Var1}$$

$$\frac{}{\langle \sigma, Q_{\texttt{IN}}, Q_{\texttt{OUT}}, x = v \rangle \xrightarrow{1} \langle \sigma[x \leftarrow v], Q_{\texttt{IN}}, Q_{\texttt{OUT}}, \texttt{skip} \rangle}\ \textsc{L-Var2}$$

$$\frac{\langle \sigma, (\texttt{pkt}, \texttt{pt}) \rangle, e \xrightarrow{p} e'}{\langle \sigma, (\texttt{pkt}, \texttt{pt}) : Q_{\texttt{IN}}, Q_{\texttt{OUT}}, \texttt{pkt}.f = e \rangle \xrightarrow{p} \langle \sigma, (\texttt{pkt}, \texttt{pt}) : Q_{\texttt{IN}}, Q_{\texttt{OUT}}, \texttt{pkt}.f = e' \rangle}\ \textsc{L-Field1}$$

$$\frac{}{\langle \sigma, (\texttt{pkt}, \texttt{pt}) : Q_{\texttt{IN}}, Q_{\texttt{OUT}}, \texttt{pkt}.f = v \rangle \xrightarrow{p} \langle \sigma, (\texttt{pkt}[f \leftarrow v], \texttt{pt}) : Q_{\texttt{IN}}, Q_{\texttt{OUT}}, \texttt{skip} \rangle}\ \textsc{L-Field2}$$

$$\frac{\langle \sigma, (\texttt{pkt}, \texttt{pt}) \rangle, e \xrightarrow{p} e'}{\langle \sigma, (\texttt{pkt}, \texttt{pt}) : Q_{\texttt{IN}}, Q_{\texttt{OUT}}, \texttt{observe}(e) \rangle \xrightarrow{p} \langle \sigma, (\texttt{pkt}, \texttt{pt}) : Q_{\texttt{IN}}, Q_{\texttt{OUT}}, \texttt{observe}(e') \rangle}\ \textsc{L-Observe1}$$

$$\frac{}{\langle \sigma, Q_{\texttt{IN}}, Q_{\texttt{OUT}}, \texttt{observe}(\mathrm{true}) \rangle \xrightarrow{1} \langle \sigma, Q_{\texttt{IN}}, Q_{\texttt{OUT}}, \texttt{skip} \rangle}\ \textsc{L-ObserveT}$$

$$\frac{\langle \sigma, (\texttt{pkt}, \texttt{pt}) \rangle, e \xrightarrow{p} e'}{\langle \sigma, (\texttt{pkt}, \texttt{pt}) : Q_{\texttt{IN}}, Q_{\texttt{OUT}}, \texttt{assert}(e) \rangle \xrightarrow{p} \langle \sigma, (\texttt{pkt}, \texttt{pt}) : Q_{\texttt{IN}}, Q_{\texttt{OUT}}, \texttt{assert}(e') \rangle}\ \textsc{L-Assert1}$$

$$\frac{}{\langle \sigma, Q_{\texttt{IN}}, Q_{\texttt{OUT}}, \texttt{assert}(\mathrm{true}) \rangle \xrightarrow{1} \langle \sigma, Q_{\texttt{IN}}, Q_{\texttt{OUT}}, \texttt{skip} \rangle}\ \textsc{L-AssertT}$$

$$\frac{}{\langle \sigma, Q_{\texttt{IN}}, Q_{\texttt{OUT}}, \texttt{assert}(\mathrm{false}) \rangle \xrightarrow{1} \langle \bot \rangle}\ \textsc{L-AssertF}$$

$$\frac{\langle \sigma, Q_{\texttt{IN}}, Q_{\texttt{OUT}}, s_1 \rangle \xrightarrow{p} \langle \sigma', Q'_{\texttt{IN}}, Q'_{\texttt{OUT}}, s'_1 \rangle}{\langle \sigma, Q_{\texttt{IN}}, Q_{\texttt{OUT}}, s_1; s_2 \rangle \xrightarrow{p} \langle \sigma', Q'_{\texttt{IN}}, Q'_{\texttt{OUT}}, s'_1; s_2 \rangle}\ \textsc{L-Seq-1}$$

$$\frac{}{\langle \sigma, Q_{\texttt{IN}}, Q_{\texttt{OUT}}, \texttt{skip}; s_2 \rangle \xrightarrow{p} \langle \sigma, Q_{\texttt{IN}}, Q_{\texttt{OUT}}, s_2 \rangle}\ \textsc{L-Seq-2}$$

$$\frac{\langle \sigma, (\texttt{pkt}, \texttt{pt}) \rangle, b \xrightarrow{p} b'}{\langle \sigma, (\texttt{pkt}, \texttt{pt}) : Q_{\texttt{IN}}, Q_{\texttt{OUT}}, \texttt{if}\ b \{ s_1 \} \texttt{else} \{ s_2 \} \rangle \xrightarrow{p} \langle \sigma, (\texttt{pkt}, \texttt{pt}) : Q_{\texttt{IN}}, Q_{\texttt{OUT}}, \texttt{if}\ b' \{ s_1 \} \texttt{else} \{ s_2 \} \rangle}\ \textsc{L-If1}$$

$$\frac{}{\langle \sigma, Q_{\texttt{IN}}, Q_{\texttt{OUT}}, \texttt{if}\ \mathrm{true} \{ s_1 \} \texttt{else} \{ s_2 \} \rangle \xrightarrow{1} \langle \sigma, Q_{\texttt{IN}}, Q_{\texttt{OUT}}, s_1 \rangle}\ \textsc{L-IfT}$$

$$\frac{}{\langle \sigma, Q_{\texttt{IN}}, Q_{\texttt{OUT}}, \texttt{if}\ \mathrm{false} \{ s_1 \} \texttt{else} \{ s_2 \} \rangle \xrightarrow{1} \langle \sigma, Q_{\texttt{IN}}, Q_{\texttt{OUT}}, s_2 \rangle}\ \textsc{L-IfF}$$

$$\frac{\langle \sigma, Q_{\texttt{IN}}, Q_{\texttt{OUT}}, \texttt{if}\ b \{ s; \texttt{while}\ b \{ s \} \} \texttt{else} \{ \texttt{skip} \} \rangle \xrightarrow{p} \langle \sigma', Q'_{\texttt{IN}}, Q'_{\texttt{OUT}}, s' \rangle}{\langle \sigma, Q_{\texttt{IN}}, Q_{\texttt{OUT}}, \texttt{while}\ b \{ s \} \rangle \xrightarrow{p} \langle \sigma', Q'_{\texttt{IN}}, Q'_{\texttt{OUT}}, s' \rangle}\ \textsc{L-While}$$

Figure 5. Bayonet local network node semantics

state to σ'_s is $p_s = P_s(\lambda, \sigma'_s \mid \sigma_s, C_1, \ldots, C_k)$. The conditional distribution P_s is specified as a stateful probabilistic program in Psi, the probabilistic language of Bayonet's backend. As an example, in Figure 6 we present the specification of a uniform scheduler (used in evaluation later). This scheduler creates an array of actions, appends all actions that can be executed in the current state, and returns an action selected uniformly at random. The scheduler can also keep state, introduced with the `state` declaration. To illustrate this option, we have declared a state variable `num_actions` in Figure 6 (though it is not needed to define a uniform scheduler).

```
def scheduler() state num_actions(0) {
  actions := []: (ℝ×ℝ)[];
  for i in [0..K){
    if (Q_in@i).size() > 0 {
      actions ~= (Run,i);
    }
    if (Q_out@i).size() > 0 {
      actions ~= (Fwd,i);
    }
  }
  num_actions += 1;
  return actions[uniformInt(0, actions.length - 1)];
}
```

Figure 6. A uniform scheduler

$$\frac{\begin{array}{l} s = \Pi(i) \qquad Q_{\text{IN},i} \neq \varnothing \qquad \lambda = (\text{Run}, i) \\ p_s = P_s(\lambda, \sigma_s' \mid \sigma_s, C_1, .., C_k) \\ C_i = \langle \sigma_i, Q_{\text{IN},i}, Q_{\text{OUT},i}, s \rangle \\ C_i' = \langle \sigma_i', Q_{\text{IN},i}', Q_{\text{OUT},i}', \texttt{skip} \rangle \\ C_i \xrightarrow{p_1} \langle \sigma_i'', Q_{\text{IN},i}'', Q_{\text{OUT},i}'', s'' \rangle \xrightarrow{p_2} \cdots \xrightarrow{p_n} C_i' \\ p = p_s \cdot p_1 \cdot p_2 \cdots p_n \end{array}}{(\sigma_s, .., C_i, ..) \overset{\lambda,p}{\Longrightarrow} (\sigma_s', .., C_i', ..)} \ \textbf{G-Run}$$

$$\frac{\begin{array}{l} ((i, \texttt{pt}), (j, \texttt{pt}')) \in \mathit{links} \qquad \lambda = (\text{Fwd}, i) \\ p_s = P_s(\lambda, \sigma_s' \mid \sigma_s, C_1, .., C_k) \\ C_i = \langle \sigma_i, Q_{\text{IN},i}, (\texttt{pkt}, \texttt{pt}) : Q_{\text{OUT},i}, s_i \rangle \\ C_j = \langle \sigma_j, Q_{\text{IN},j}, Q_{\text{OUT},j}, s_j \rangle \\ C_i' = \langle \sigma_i, Q_{\text{IN},i}, Q_{\text{OUT},i}, s_i \rangle \\ C_j' = \langle \sigma_j, Q_{\text{IN},j} : (\texttt{pkt}, \texttt{pt}'), Q_{\text{OUT},j}, s_j \rangle \end{array}}{(\sigma_s, .., C_i, .., C_j, ..) \overset{\lambda,p_s}{\Longrightarrow} (\sigma_s', .., C_i', .., C_j', ..)} \ \textbf{G-Fwd}$$

Figure 7. Bayonet global network semantics

The global network semantics is given as transition steps of the form $(\sigma_s, C_1, .., C_k) \overset{\lambda,p}{\Longrightarrow} (\sigma_s', C_1', .., C_k')$. We label these steps with the action λ selected by the scheduler and the probability p of the transition. In Figure 7, we define Bayonet's global network semantics. The action (Run, i) executes a number of steps of node i's Bayonet program. The probability of this step is $p = p_s \cdot p_1 \cdot p_2 \cdots p_n$, where p_s is the probability with which the scheduler selects the action (Run, i) and p_j, with $j \in [1 \ldots n]$, is the probability of the local step j executed by node i.

The action (Fwd, i) takes the packet `pkt` at port `pt` at the head of node i's output queue and forwards it to the input queue of its destination. The destination is determined by checking which node is connected across the interface $(i, \texttt{pt})$. The probability of this step equals the probability that the scheduler selects the action (Fwd, i).

(Values)	v	$\in$	$\mathit{Vals} = \mathbb{Q}$
(Vars)	x	$\in$	Vars
(Nodes)	n	$\in$	$\mathit{Nodes} = \{1, \ldots, k\}$, for $k \in \mathbb{N}$
(State expressions)	e	$::=$	$v \mid x@n \mid e + e \mid v \cdot e$
(State conditions)	b	$::=$	$e == e \mid e < e \mid b$ `and` $b \mid$ `not` b
(Query)	Q	$::=$	`expectation`$(e) \mid$ `probability`(b)

Figure 8. Syntax for specifying network properties

Network Traces Bayonet checks properties on terminal network configurations. A configuration $N = (\sigma_s, C_1, .., C_k)$ is terminal if: (i) the input and output queues of all nodes are empty and their statements are fully evaluated, or (ii) there is a node i that is in an error state due to a failed `assert`.

Given an initial configuration $N_{init} = (\sigma_s, C_1, .., C_k)$, we define a network trace as $N_{init} \overset{\lambda,p}{\Rrightarrow} N_m$ such that $N_{init} \overset{\lambda_1,p_1}{\Longrightarrow} N_1 \overset{\lambda_2,p_2}{\Longrightarrow} \cdots \overset{\lambda_m,p_m}{\Longrightarrow} N_m$, $\lambda = \lambda_1, .., \lambda_m$, $p = p_1 \cdot .. \cdot p_m$, and all intermediate network configurations $N_1, .., N_{m-1}$ are non-terminal. Here, each $\lambda_i \in \{\text{Run}, \text{Fwd}\} \times \{1, \ldots, k\}$ is an action selected by the scheduler. We denote the set of all traces for a configuration N_{init} by $\mathit{Traces}(N_{init})$. The set is finite for any configuration N_{init} that always reaches a terminal one.

Network Configuration Probabilities The unnormalized probability of reaching a network configuration N is the aggregate of the probabilities of all traces that lead to N. We define unnormalized aggregate trace semantics: $N_{init} \overset{p}{\Rrightarrow}_{\text{u}} N$, where $p = \sum p_t$ such that $t \in \mathit{Traces}(N_{init})$ and $N_{init} \overset{t,p_t}{\Rrightarrow} N$.

Let Z be the sum of unnormalized probabilities of terminal network states. Due to observation failures, Z may be lower than 1. To obtain normalized probabilities, we divide unnormalized probabilities by Z. The normalized aggregate trace semantics is then given by $N_{init} \overset{p/Z}{\Rrightarrow} N$ where $N_{init} \overset{p}{\Rrightarrow}_{\text{u}} N$.

3.3 Network Properties

Bayonet allows network operators to capture network properties by evaluating expressions and conditions on the local states of network nodes. As we show in our evaluation, this is sufficient to check many practical network properties.

We depict the syntax for specifying network properties in Figure 8. To check the state of local nodes, network operators can use the expression $x@n$ where x is a variable and n is a network node identifier. This expression is evaluated at a terminal network configuration N, and returns the value of variable x at the local state of node n. For example, `pkt_cnt@H1` returns the value of variable `pkt_cnt` according to the state of host `H1`. Using this expression, network operators can express

- *state expressions* e, such as `pkt_cnt@H1 - pkt_cnt@H0`;
- *state conditions* b, such as `pkt_cnt@H1 < 3`.

Operators can specify two kinds of Bayonet queries:

- `expectation`(e), which returns the expected value of expression e for all non-error terminating network configurations;
- `probability`(b), which returns the probability that condition b holds in all terminating network configurations.

For example, the query `expectation`(`pkt_cnt@H1`) returns the expected number of packets (`pkt_cnt`) received by host `H1`, and `probability`(`pkt_cnt@H1 < 3`) returns the probability that host `H1` receives less than three packets.

4 Implementation

Bayonet supports inference with two probabilistic inference solvers: (i) Psi, which performs exact symbolic inference, and (ii) WebPPL, which performs approximate inference. To leverage these solvers, the Bayonet system translates Bayonet programs to a program in the source language of Psi, which then has the options to either use its exact inference engine or to translate to WebPPL programs.

Both inference solvers can be used to compute a concrete value for a given probability/expectation query. Parameter synthesis is supported only by the exact solver. It can output a symbolic expression parameterized by network parameters (which can be instantiated to concrete values). The Bayonet system is available at: http://bayonet.ethz.ch/.

To provide an intuition for our translation process, we illustrate the translation of Bayonet programs to Psi and then show how the global network semantics is encoded as a Psi program. The entire process (translation step) is fully automated.

Translating* Bayonet *programs to Psi Figure 9 presents the Bayonet program of switch `S0` (top) taken from our network example in Section 2, and the translated version of this program in Psi (bottom). The translator declares two queues `Q_in` and `Q_out` to capture the input and, respectively, output queues of the network node. The constructor `s0` initializes the queues and the state variables `route1` and `route2`.

The semantics of the Bayonet program `s0` is captured by the method `run()`. This method takes the pair `(pkt, pt)` from the input queue `Q_in`, capturing that statements are applied to the packet and port on top of the input queue. To capture rule S-Fwd, Bayonet translates the statements `fwd(pt`)` to `Q_out.pushBack(pkt, pt`)`, which places the current packet and the new port `pt`` at the back of the output queue `Q_out`.

Capturing Network Semantics In addition to translating Bayonet programs of hosts and switches, the system also generates a Psi program that captures the global network semantics. The generated Psi program for our network example is given in Figure 10.

The map `programs` associates network nodes to Bayonet programs and the map `links` captures the topology of the network. The method `scheduler()` defines the behavior of the scheduler; see Figure 6 for an example. The method `step()` captures the global network steps. In each step, the network executes a program at the node if the scheduler returns the

```
def s0(pkt, pt) state route1(0), route2(0){
  if pt == 1 {
    fwd(3);
  } else if pt == 2 {
    if pkt.dst == H0 {
      fwd(3);
    } else {
      fwd(1);
    }
  } else if pt == 3 {
    route1 = COST_01;
    route2 = COST_02 + COST_21;
    if route1 < route2 or
       (route1 == route2 and flip(1/2)) {
      fwd(1);
    } else {
      fwd(2);
    }
  }
}
```

(a) Bayonet program of switch `S0`

```
dat s0{
  Q_in: Queue, Q_out: Queue;
  route1: ℝ, route2: ℝ;
  def s0() { // (constructor)
    Q_in = Queue();
    Q_out = Queue();
    route1 = 0;
    route2 = 0;
  }
  def run() {
    (pkt, pt) := Q_in.takeFront();
    if pt == 1 {
      Q_out.pushBack(pkt, 3);
    } else if pt == 2 {
      if pkt.dst == H0 {
        Q_out.pushBack(pkt, 3);
      } else {
        Q_out.pushBack(pkt, 1);
      }
    } else if pt == 3 {
      route1 = COST_01;
      route2 = COST_02 + COST_21;
      if route1 < route2 ||
        (route1 == route2 && flip(1/2)) {
        Q_out.pushBack(pkt, 1);
      } else {
        Q_out.pushBack(pkt, 2);
} } } } } }
```

(b) Translation to Psi

Figure 9. Translating a Bayonet program to Psi

```
1  dat Network {
2    programs := [ 0 ↦ h0(), 1 ↦ s0(), ..];
3    links := [ (0, 1) ↦ (1, 3), ..];
4    scheduler_state := ...;
5    def scheduler() {..}
6    def step(){
7      (action, node_id) := scheduler();
8      if action == Run {
9        programs[node_id].run();
10     }
11     if action == Fwd {
12       (pkt, out_pt) :=
13         programs[node_id].Q_out.takeFront();
14       (dst_id, dst_pt) := links[(node_id, out_pt)];
15       programs[dst_id].Q_in.pushBack(pkt, dst_pt);
16   } }
17   def terminated()⇒...;
18   def main() {
19     repeat num_steps {
20       if !terminated() {
21         step();
22       }
23     }
24     assert(terminated());
25     return (<query>); // <query> to check
26 } }
```

Figure 10. Capturing the global network semantics

action Run, and otherwise it forwards a packet from a node's output queue to another node's input queue.

Finally, the method `main()` unrolls the steps performed by the network. The method `step()` is called whenever the network has not reached a terminal state. This is checked via a call to method `terminated()`, which evaluates the terminal-state condition defined in Section 3.2. The `assert` statement at line 24 checks that the given number of steps is sufficient, such that the network reaches a terminal state. The program then returns a concrete result to the Bayonet query (i.e., an `expectation` or a `probability` query). If the operator leaves some of the input network parameters symbolic, then the Bayonet system outputs a symbolic expression that captures the possible values of the Bayonet query in terms of these symbolic parameters.

Integrity Checking Before translating programs, Bayonet checks statically for several common problems when defining networks. The checks include that each defined node is used and assigned a proper program, that all nodes are linked and that each port is connected to only one link, that the queue capacities are non-negative, that there is at least one query declaration and that the number of steps for which to simulate the network is specified exactly once. These checks are domain-specific and were easy to implement for Bayonet programs.

Benefits of Translation Our experience is that the translation of Bayonet programs into Psi programs is fairly direct. In particular, Bayonet's operational semantics specifies how to implement an interpreter for the language, while Psi's expressiveness is a good match for the back-end language. This also interacts well with the ability of Psi to symbolically analyze probabilistic programs. Similarly, the translation to WebPPL is also direct (although using a different set of language primitives). Note that Bayonet can use any solver that is powerful enough to capture its semantics. Support for further solvers can be added in the future. Bayonet can provide well-motivated benchmark problems for those solvers.

We note that the resulting translated programs often have many more lines of code and are harder to read than Bayonet programs. In our experiments, Bayonet programs are two to ten times smaller than the corresponding generated PSI or WebPPL programs.

The translation approach supports the hypothesis of this paper, namely, that existing probabilistic languages are a suitable backend for more restricted domain-specific languages (DSLs), yet such DSLs are more suitable and convenient for domain experts to work with than general probabilistic languages. An interesting future work item is formally proving the correctness of such translations.

Complexity Exact inference for an arbitrary Bayonet program is a hard problem, because the Bayonet language can encode non-trivial semantic properties of Turing machines whose running time can be exponential in the number of characters of the Bayonet program. In practice, the performance of exact inference will depend on the size of the state space, as well as on the capabilities of the exact solver backend to exploit structure in the given problem instance (such that it does not need to explicitly visit all reachable program states). The size of the state space depends on the number of network nodes, links, packets, queue capacities as well as other specifics such as the exact forwarding rules used, how many executions are cut short by failing assertions and observations, and whether packets are distinguishable. In general, computing the size of the state space for a given problem instance is an interesting problem in itself. For example, in a fully-connected network topology on n nodes with k distinguishable packets and queue capacity c such that any distribution of packets into queues can be reached, there are $\sum_{i=\min(c,k)}^{k} i! \cdot (n-1)^i \cdot [x^i](\sum_{j=1}^{c} x^j)^{2n}$ states ($[x^i]p(x)$ denotes the coefficient of x^i in a polynomial p).

For approximate inference, the size of the state space is less important. For programs without observations, the probability of a property can be estimated with good probabilistic guarantees using standard sampling.

If the property is very unlikely, or the program is conditioned on unlikely events (as it often happens for networks,

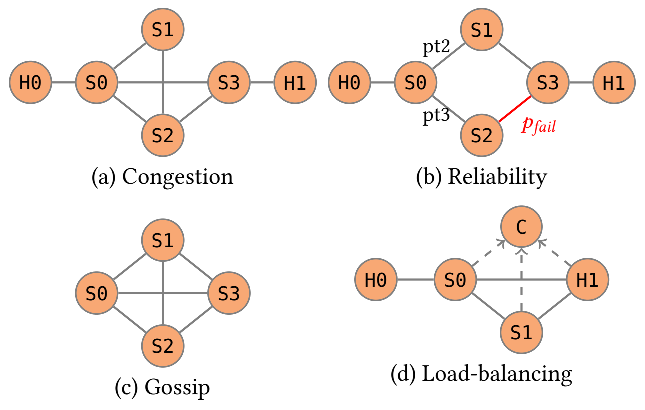


Figure 11. Network topologies used in our benchmarks

e.g. any specific sequence of observed packets may be unlikely), accurate approximate inference becomes more challenging. To us, a combination of exact and approximate inference approaches seems to be the most promising approach for solving such instances.

As Bayonet is decoupled from the inference algorithms employed by its backends, it will automatically benefit as those algorithms improve over time.

5 Evaluation

In this section we present a detailed evaluation of our approach on a range of practically-relevant network scenarios.

Benchmarks We considered examples that were previously studied in the context of manual network verification [22] but also other queries, beyond those handled in prior work, including probabilistic inference using evidence and parameter synthesis.

We model several network topologies and properties: (i) probability of congestion in networks with probabilistic routing; (ii) reliability of packet delivery in the presence of probabilistic failures; (iii) expected message propagation for gossip protocols; (iv) Bayesian reasoning using observations for conformance with load-balancing policies and congestion control; and (v) parameter synthesis for congestion (described in Section 2.3). Full code for all benchmarks is available at: http://bayonet.ethz.ch/.

We used the network topologies shown in Figure 11. To showcase the scalability of Bayonet when using approximate inference, we scaled topology (b) to 30 nodes for congestion and reliability, by connecting multiple copies of topology (b) in a row, and scaled the fully-connected topology (c) to 30 nodes for gossip. Note that 30 nodes is a practical network size: a recent analysis of 141 production networks reported that 70% of them have 30 nodes or less [39].

For each benchmark, we analyzed at least one representative query (which we describe in each benchmark's section). We used two different schedulers: (i) a uniform probabilistic scheduler described on page 8, and (ii) a deterministic scheduler that does not use random choices. For approximate inference in WebPPL, we used the Sequential Monte Carlo inference method (SMC) with 1000 particles.

Benchmark	**Sched.**	**Nodes**	**Exact**		**Approximate**	
			Result	*Time*	*Result*	*Time*
Congestion	uni.	5	0.4487	65s	0.4570	22s
Congestion	det.	5	1.0000	0.5s	1.0000	19s
Congestion	uni.	6	0.4441	595s	0.4650	29s
Congestion	det.	6	1.0000	1.4s	1.0000	21s
Congestion	det.	30	1.0000	311s	1.0000	320s
Reliability	uni.	6	0.9995	0.2s	0.9990	11s
Reliability	det.	6	0.9995	0.2s	1.0000	10s
Reliability	uni.	30	0.9965	192s	0.9940	288s
Reliability	det.	30	0.9965	240s	0.9980	304s
Gossip	uni.	4	3.4815	362s	3.4760	11s
Gossip	det.	4	3.4815	12s	3.4890	11s
Gossip	uni.	20	–	-	16.0020	312s
Gossip	uni.	30	–	-	23.9910	879s

Table 1. Bayonet results for our benchmarks

All benchmarks were run on a AMD EPYC 7601 Processor with 500 GB RAM.

Questions We aim to answer three research questions:

- *Can* Bayonet *capture interesting network scenarios succinctly?* To answer this question we present our implementation of the problems, including the relevant code snippets. We also discuss how the results depend on the design choices such as schedulers.
- *Can* Bayonet *perform an efficient inference for networks of realistic size?* To answer this question, we run both the exact and approximate inference for networks up to size 30, a representative size for many real networks (see above).
- *Can* Bayonet *perform effective inference with observations?* To answer this question, we focus on two problems: load balancing and reliability analysis.

For each benchmark, Table 1 presents, for both the exact (Psi) and the approximate (WebPPL) inference engines, the result of the query and inference times. We also note that the code size of Bayonet is significantly smaller than the code generated for the other languages: around 50% less than Psi and around 10x less than WebPPL. In the following, we describe the benchmarks individually. We discuss scaling in Section 5.4 and inference with observations in Section 5.5.

5.1 Network Congestion for Probabilistic Routing

Network congestion can seriously degrade a network's performance and utilization. Knowing the probability of congestion is therefore of great interest to network operators.

We measure the probability of congestion for: (i) the network example presented in Section 2 and (ii) the network shown in Figure 11(a). In both networks, routers direct traffic

```
def h1(pkt, port)
  state arrived(0) {
  arrived=1;
  drop;
}

def s0(pkt, port) {
  if flip(1/2) {
    fwd(2); // to S1
  } else {
    fwd(3); // to S2
  }
}
```

```
def s2(pkt,port)
  state failing(2) {
  if failing == 2 {
    failing =
      flip(P_FAIL);
  }
  if failing == 1 {
    drop;
  } else{
    fwd(2); // to S3
  }
}
```

Figure 12. Bayonet programs for our reliability benchmark

along the least-cost paths and they load balance traffic using ECMP routing if there are multiple equal-cost paths. The link costs are such that all paths between host nodes H_0 and H_1 have the same cost. The capacity of all routers' queues is 2. The host H_0 sends three packets to H_1. The probability of congestion is thus given by `probability(pkt_cnt@H1 < 3)`.

In Table 1 (first five rows) we show the query results for this experiment. Congestion probabilities computed by both inference engines are similar: they are identical for the deterministic scheduler and the difference across the two methods for the uniform scheduler is < 0.03. Note that the specific deterministic round-robin scheduler used for this experiment considers only runs in which congestion occurs. This hides important information, because in practice it is likely that a different interleaving occurs. This is modeled better by the uniform scheduler.

5.2 Reliability of Packet Delivery

Reliability of packet delivery is defined as the probability that a given packet (or, a flow of packets) is delivered to the intended destination. Service-level agreements, such as "*99% of all packets destined to network `10.0.1/24` are delivered.*", are often formulated in terms of reliability requirements. Calculating the probability of packet delivery is thus important.

We consider a diamond network topology (Figure 11(b)) with a single link (colored in red) that fails with probability $p_{fail} = 1/1000$. Host H_0 sends a packet to host H_1. Switch S_0 uses the probabilistic ECMP forwarding strategy, and thus forwards half of the H_0's packets to S_1 and the other half to S_2. Switch S_2 forwards packets to S_3 unless the link between S_2 and S_3 has failed, in which case S_2 drops all packets.

The Bayonet programs of switches S_0 and S_2 and of host H_1 are given in Figure 12. The variable `arrived` is set to 1 whenever H_1 receives a packet. We can capture the reliability of packet delivery with the query `probability(arrrived@H1)`.

The results for this query are given in Table 1 (rows 6-8). The scheduler does not influence the query result, because in this experiment we track a single packet. The results computed using the approximate method are close (± 0.005) to those computed by the exact one.

5.3 Expected Message Propagation for Gossip Protocols

Gossip protocols are used to spread information among nodes in a randomized fashion. In each round, a node selects a neighbor at random and shares a piece of information. Gossip protocols have many practical applications. For example, Google's Certificate Transparency project uses them to spread information about suspicious SSL certificates. An important query is to compute the expected number of nodes that received the information after a given number rounds.

We consider the network topology depicted in Figure 11(c). Node S_0 starts the process by becoming infected and sending a single packet to a random adjacent node. Each uninfected node that receives a packet becomes infected and sends two more packets to (not necessarily distinct) random adjacent nodes. The goal is to analyze the distribution of the number of nodes that will become infected in total.

In the Bayonet program of a node S_i, we introduce a variable `infected@Si` to represent whether it is infected. We capture the expected number of infected nodes with the query `expectation`$(\sum_i$ `infected@`$S_i)$. The expected number of infected nodes, computed with exact inference, is $94/27 \approx 3.4815$ (for both schedulers); see Table 1. As before, the approximate procedure computes results that are close to the exact one.

5.4 Performance and Network Size

To test scalability of Bayonet's exact and approximate inference, we analyzed the computation with networks of a larger size (up to 30). Table 1 presents the performance of both exact and approximate inferences.

For congestion and reliability, both exact and approximate inference can produce results of similar accuracy within 8 minutes. For gossip, approximate inference can produce results within 11 minutes. The exact method did not terminate within an hour for these queries.

Overall, the performance results show that Bayonet is effective in analyzing non-trivial networks. Moreover, the fact that the exact solver is faster than the approximate solver for some benchmarks emphasizes the need for practical networking languages to use multiple backends.

5.5 Bayesian Reasoning using Observations

We next discuss several practical network scenarios which involve Bayesian reasoning, supported by our approach.

Probability of Correct Load-balancing Switches that use ECMP routing compute a hash of the packet header to decide the link for forwarding the packet. Operators select a hash

```
def h1()
  state num_arr(0){
  num_arr = num_arr+1;
  if num_arr==1 {
    observe(pkt.id==1);
  } else if num_arr==2 {
    observe(pkt.id==3);
  } else if num_arr==3 {
    observe(0);
  }
  drop;
}
```

(a) Observation $(1, 3)$

```
def h1()
  state num_arr(0){
  num_arr = num_arr+1;
  if num_arr==1 {
    observe(pkt.id==1);
  } else if num_arr==2 {
    observe(pkt.id==2);
  } else if num_arr== 3 {
    observe(pkt.id==3);
  }
  drop;
}
```

(b) Observation $(1, 2, 3)$

Figure 13. Observations added to the program of host H_1

function to split traffic across all outgoing links as prescribed by a load-balancing policy.

We consider the network shown in Figure 11(d). Switch S_0 receives traffic from H_0 and splits it, using ECMP, among a direct link to H_1 and a direct link to S_1. Switch S_1 forwards all incoming traffic to H_1. S_0, S_1 and H_1 are connected to a controller C, which monitors the network. There is a fixed sub-sampling probability: each node connected to the controller chooses randomly whether or not to send a copy of the packet to the controller. This way, the controller gets an incomplete, but representative picture of the network traffic.

We use Bayonet to perform Bayesian reasoning, using observations, to compute the probability that S_0's hash function is bad (captured by `probability(bad_hash@S1)`). We first choose a *prior probability* on two models of S_0's forwarding behavior. We believe it is likely (probability $9/10$) that S_0 splits the traffic equally, forwarding a packet to host H_1 with probability $1/2$. However, we are aware of the less likely scenario (probability $1/10$) where S_0's hash function is bad, forwarding a packet to host H_1 with a probability of only $1/3$. The prior is therefore a Bernoulli$(1/10)$ random variable specifying whether the hash is bad. This prior is encoded as a simple *generative model* and is part of the Bayonet program: `def s1() state bad_hash(flip(1/10)){..}`.

Host H_0 sends three packets to H_1. S_0, S_1, and H_1 randomly choose whether to send a copy of these packets to the controller. We perform two experiments: in the first experiment, the controller observes packets from S_1, S_0, S_0, S_1, H_1, in that order; in the second experiment, the observed packets are from H_1, S_0, S_0, H_1. The sequence of observations in the first experiments hints at a bad hash function, while in the second experiment it is indicative of a good hash function. Using this information, Bayonet automatically updates the prior to compute the *posterior probability* that S_0's hash function is bad. As expected, the probability that the hash function is bad increases in the first experiment, to 0.152, indicating a bad hash function, and it decreases in the second one, to 0.004, indicating a good hash function.

Reliability of Packet Delivery We give another example that illustrates how Bayonet is used to perform Bayesian reasoning using observations.

We adapt the reliability scenario from Section 5.2 as follows. We consider a setting where we are uncertain of the forwarding strategy implemented by switch S_0. A priori, we consider two equally likely hypotheses: (i) S_0 selects a uniform random host out of $\{S_1, S_2\}$ for each packet; we denote this strategy by *rand*, (ii) S_0 forwards all packets to the same host. If case (ii) applies, we also consider it equally likely that S_0 always forwards to S_1 and that S_0 always forwards to S_2; we denote these strategies by *det.* S_1 and *det.* S_2, respectively.

We consider the scenario where we can observe the packets arriving at host H_1. We know that three packets, numbered $1, 2, 3$, have been sent from host H_0, and we have observed an (exhaustive) sequence of packets arriving at host H_1. We can use the observed sequence of packets arrived at H_1 to refine our knowledge of S_0's forwarding strategy.

To illustrate our analysis, we consider two example observations: (i) packet 1 arrives before packet 3, while packet 2 does not arrive at all; and (ii) all three packets arrived in the same order they were sent. These two scenarios can be expressed by modifying the program of H_1 (Figure 13).

For the observation $(1, 3)$, the posterior distribution output by the exact inference of Bayonet is: $\Pr(\text{rand}) = 1$, $\Pr(\text{det. } S_1) = 0$, and $\Pr(\text{det. } S_2) = 0$. The result is sensible: the only way that the data can be explained is if S_0 forwarded packets 1 and 3 to switch S_1, and packet 2 to switch S_2. The link from S_2 to S_3 failed, such that packet 2 was dropped.

For the observation $(1, 2, 3)$, we get the following posterior:

$$\Pr(\text{rand}) = 41922792469/95643630613 \approx 0.4383,$$
$$\Pr(\text{det. } S_1) = 26873856000/95643630613 \approx 0.2810,$$
$$\Pr(\text{det. } S_2) = 26846982144/95643630613 \approx 0.2807.$$

The posterior reflects that random forwarding is less likely than deterministic forwarding. This is because the packets were not reordered, and for the random forwarding strategy, the packets are likely to be received out of order.

Further, forwarding deterministically to S_1 is slightly more likely than to S_2. The reason is that there is a small chance the link between S_2 and S_3 will fail, in which case no packets would arrive when deterministically forwarding to S_2.

6 Related Work

We now discuss work that is most closely related to ours.

ProbNetKAT We start by comparing to ProbNetKAT [22] (PNK) and its recent automation [61]:

- *Automated Inference*: PNK's approach to automation is to provide a custom analyzer for programs in its language [61], while Bayonet translates to standard probabilistic programs. This allows Bayonet to leverage a rich body of existing work on probabilistic inference. Further, PNK does not support parameter synthesis.

- *Synchronous vs. Asynchronous*: PNK is based on a synchronous model, while Bayonet uses a stateful probabilistic scheduler. As networks are asynchronous, we believe Bayonet allows for more realistic modeling.
- *Observations*: In contrast to Bayonet, PNK does not support `observe` statements. While it may be possible to extend PNK with `observe` statements, doing so requires care to avoid issues: a direct addition of `observe` as in [40] can indeed invalidate the main theorem of [61].
- *History vs. State*: Bayonet models state (e.g., packet queues), while PNK does not and uses sets of packet histories. The use of state in Bayonet strikes a good balance between expressiveness and efficiency of analysis: it can capture properties such as loops, congestion, and reliability [18, 36, 42, 51] as these do not need histories. The use of state instead of histories also greatly reduces the burden on underlying probabilistic solvers. On the other side, histories allow to elegantly capture temporal properties that will require auxiliary state in Bayonet.
- *Other Differences*: PNK has a limited support for conditionals (e.g., comparing packet fields only to constants, not to other fields) and computations (e.g., cannot count the number of hops in a packet). In contrast, Bayonet supports standard conditionals, arithmetic, and assignments.

Probabilistic Programming Recent years have seen significant interest in probabilistic programming languages including Stan [25], PSI [24], Fabular [8], Anglican [67], Augur [63], Church [26], Venture [43] and R2 [54]. Uncertain<T> is a library that makes probabilistic programming features available inside mainstream programming languages [9, 50].

The main purpose of these systems is to simplify the development of probabilistic models. The systems support different methods for probabilistic inference, including translation to Bayesian networks (e.g., [54]), sampling (e.g., [67]), exact symbolic inference for bounded length executions with both discrete and continuous distributions [24, 52, 60], probabilistic abstract interpretation [44, 45], axiomatic methods [5, 35, 48], analysis based on probabilistic model checking [33], and other methods [6, 7, 14, 20, 31, 59]. For a survey on probabilistic programming, please see [28].

Researchers in computer networks have used Bayesian inference for problems in traffic classification [4, 15, 47, 66], security [23], and network management [34]. However, these solutions use specific inference techniques, manually tailored for a specific problem. As such, they lack generality and cannot benefit from the latest advances in the field of probabilistic inference. Also, many of the previous techniques tackle problems in data analysis rather than network-wide reasoning. In contrast, by connecting probabilistic network reasoning to standard probabilistic programming, Bayonet can directly leverage the significant effort spent in developing advanced inference algorithms and systems and benefit from recent and future advances in probabilistic analysis [1, 2, 31, 50], compilation [30, 68], and synthesis [12, 55].

Comparison with Network Simulators Network simulators [56, 64, 65] or emulators [11, 41] can typically operate in deterministic mode where every run produces the same result, and randomized mode, where a run produces a different result, sampled from a (uniform) pseudorandom number generator. Statistical model checkers use similar approach to refine non-deterministic into probabilistic choice [16]. Some simulators expose probabilistic choice to the users. For instance, in the NS2 network simulator, packets can have random content and the scheduler can select the next task uniformly at random. A user can sample data from standard distributions and specify potentially randomized behavior of the scheduler using C code [32]. To mirror this flexibility in specifying probabilistic behaviors, Bayonet allows a developer to use various standard distributions and randomness in data, programs, and schedulers.

Network Analysis Over the last few years there has been substantial interest in analysis and verification of network behaviors, both usual network protocols and software defined networking (SDN), e.g., [10, 19, 21, 38, 62]. These efforts are largely orthogonal to our work as they do not consider probabilistic behaviors.

7 Conclusion

We presented Bayonet, a novel approach for expressing and reasoning about probabilistic networks. Our approach consists of two parts: a probabilistic language capable of expressing practical network scenarios together with an automated system that performs probabilistic reasoning. A key idea behind Bayonet's system is to phrase the problem of probabilistic network reasoning as classic probabilistic inference, accomplished by compiling Bayonet programs to standard probabilistic programs. An important benefit of this approach is that Bayonet brings state-of-the-art advances in probabilistic inference (e.g., Bayesian reasoning) to the domain of networks. We demonstrated that Bayonet can express interesting and practical network scenarios and automatically reason about useful probabilistic network properties. Based on these results, we believe that Bayonet is a solid basis for reasoning about probabilistic networks.

Acknowledgements

We thank our shepherd Jean-Baptiste Tristan, as well as Benjamin Bichsel, Keyur Joshi, and the anonymous reviewers for their useful comments on drafts of the paper. This research was supported in part by SNF (grant number 163117) and NSF (Grants No. CCF 1703637 and CCF 1629431).

References

[1] Aws Albarghouthi, Loris D'Antoni, Samuel Drews, and Aditya V. Nori. 2017. FairSquare: Probabilistic Verification of Program Fairness. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 80 (Oct. 2017), 30 pages. https://doi.org/10.1145/3133904

[2] Torben Amtoft and Anindya Banerjee. 2016. A Theory of Slicing for Probabilistic Control Flow Graphs. In *Foundations of Software Science and Computation Structures*, Bart Jacobs and Christof Löding (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 180–196. https://doi.org/10.1007/978-3-662-49630-5_11

[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-SIGACT Symposium on Principles of Programming Languages (POPL '14)*. ACM, New York, NY, USA, 113–126. https://doi.org/10.1145/2535838.2535862

[4] T. Auld, A. W. Moore, and S. F. Gull. 2007. Bayesian Neural Networks for Internet Traffic Classification. *IEEE Transactions on Neural Networks* 18, 1 (Jan 2007), 223–239. https://doi.org/10.1109/TNN.2006.883010

[5] Gilles Barthe, Boris Köpf, Federico Olmedo, and Santiago Zanella-Béguelin. 2013. Probabilistic Relational Reasoning for Differential Privacy. *ACM Trans. Program. Lang. Syst.* 35, 3, Article 9 (Nov. 2013), 49 pages. https://doi.org/10.1145/2492061

[6] Sooraj Bhat, Ashish Agarwal, Richard Vuduc, and Alexander Gray. 2012. A Type Theory for Probability Density Functions. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '12)*. ACM, New York, NY, USA, 545–556. https://doi.org/10.1145/2103656.2103721

[7] Sooraj Bhat, Johannes Borgström, Andrew D. Gordon, and Claudio Russo. 2013. Deriving Probability Density Functions from Probabilistic Functional Programs. In *Tools and Algorithms for the Construction and Analysis of Systems*, Nir Piterman and Scott A. Smolka (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 508–522. https://doi.org/10.1007/978-3-642-36742-7_35

[8] Johannes Borgström, Andrew D. Gordon, Long Ouyang, Claudio Russo, Adam Ścibior, and Marcin Szymczak. 2016. Fabular: Regression Formulas As Probabilistic Programming. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16)*. ACM, New York, NY, USA, 271–283. https://doi.org/10.1145/2837614.2837653

[9] James Bornholt, Todd Mytkowicz, and Kathryn S. McKinley. 2014. Uncertain<T>: A First-order Type for Uncertain Data. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '14)*. ACM, New York, NY, USA, 51–66. https://doi.org/10.1145/2541940.2541958

[10] Marco Canini, Daniele Venzano, Peter Perešíni, Dejan Kostić, and Jennifer Rexford. 2012. A NICE Way to Test Openflow Applications. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI'12)*. USENIX Association, Berkeley, CA, USA, 10–10. http://dl.acm.org/citation.cfm?id=2228298.2228312

[11] Marta Carbone and Luigi Rizzo. 2010. Dummynet Revisited. *SIGCOMM Comput. Commun. Rev.* 40, 2 (April 2010), 12–20. https://doi.org/10.1145/1764873.1764876

[12] Sarah Chasins and Phitchaya Mangpo Phothilimthana. 2017. Data-Driven Synthesis of Full Probabilistic Programs. In *Computer Aided Verification*, Rupak Majumdar and Viktor Kunčak (Eds.). Springer International Publishing, Cham, 279–304.

[13] B. Claise. 2004. Cisco Systems NetFlow Services Export Version 9. RFC 3954 (Informational). (Oct. 2004). http://www.ietf.org/rfc/rfc3954.txt

[14] Guillaume Claret, Sriram K. Rajamani, Aditya V. Nori, Andrew D. Gordon, and Johannes Borgström. 2013. Bayesian Inference Using Data Flow Analysis. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2013)*. ACM, New York, NY, USA, 92–102. https://doi.org/10.1145/2491411.2491423

[15] A. Dainotti, W. de Donato, A. Pescape, and P. Salvo Rossi. 2008. Classification of Network Traffic via Packet-Level Hidden Markov Models. In *IEEE GLOBECOM 2008 - 2008 IEEE Global Telecommunications Conference*. 1–5. https://doi.org/10.1109/GLOCOM.2008.ECP.412

[16] Alexandre David, Kim G. Larsen, Axel Legay, Marius Mikučionis, and Danny Bøgsted Poulsen. 2015. Uppaal SMC tutorial. *International Journal on Software Tools for Technology Transfer* 17, 4 (01 Aug 2015), 397–415. https://doi.org/10.1007/s10009-014-0361-y

[17] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

[18] Oscar Diaz, Feng Xu, Nasro Min-Allah, Mahmoud Khodeir, Min Peng, Samee Khan, and Nasir Ghani. 2012. Network Survivability for Multiple Probabilistic Failures. *IEEE Communications Letters* 16, 8 (August 2012), 1320–1323. https://doi.org/10.1109/LCOMM.2012.060112.120353

[19] Ahmed El-Hassany, Jeremie Miserez, Pavol Bielik, Laurent Vanbever, and Martin Vechev. 2016. SDNRacer: Concurrency Analysis for Software-defined Networks. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16)*. ACM, New York, NY, USA, 402–415. https://doi.org/10.1145/2908080.2908124

[20] Antonio Filieri, Corina S. Păsăreanu, and Willem Visser. 2013. Reliability analysis in Symbolic PathFinder. In *2013 35th International Conference on Software Engineering (ICSE)*. 622–631. https://doi.org/10.1109/ICSE.2013.6606608

[21] Ari Fogel, Stanley Fung, Luis Pedrosa, Meg Walraed-Sullivan, Ramesh Govindan, Ratul Mahajan, and Todd Millstein. 2015. A General Approach to Network Configuration Analysis. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation (NSDI'15)*. USENIX Association, Berkeley, CA, USA, 469–483. http://dl.acm.org/citation.cfm?id=2789770.2789803

[22] Nate Foster, Dexter Kozen, Konstantinos Mamouras, Mark Reitblatt, and Alexandra Silva. 2016. Probabilistic NetKAT. In *Programming Languages and Systems*, Peter Thiemann (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 282–309. https://doi.org/10.1007/978-3-662-49498-1_12

[23] P. García-Teodoro, J. Díaz-Verdejo, G. Maciá-Fernández, and E. Vázquez. 2009. Anomaly-based Network Intrusion Detection: Techniques, Systems and Challenges. *Computers & Security* 28, 1-2 (Feb. 2009), 18–28. https://doi.org/10.1016/j.cose.2008.08.003

[24] Timon Gehr, Sasa Misailovic, and Martin Vechev. 2016. PSI: Exact Symbolic Inference for Probabilistic Programs. In *Computer Aided Verification*, Swarat Chaudhuri and Azadeh Farzan (Eds.). Springer International Publishing, Cham, 62–83. https://doi.org/10.1007/978-3-319-41528-4_4

[25] Andrew Gelman, Daniel Lee, and Jiqiang Guo. 2015. Stan: A Probabilistic Programming Language for Bayesian Inference and Optimization. *Journal of Educational and Behavioral Statistics* 40, 5 (oct 2015), 530–543. https://doi.org/10.3102/1076998615606113

[26] Noah D. Goodman, Vikash K. Mansinghka, Daniel Roy, Keith Bonawitz, and Joshua B. Tenenbaum. 2008. Church: A Language for Generative Models. In *Proceedings of the Twenty-Fourth Conference on Uncertainty in Artificial Intelligence (UAI'08)*. AUAI Press, Arlington, Virginia, United States, 220–229. http://dl.acm.org/citation.cfm?id=3023476.3023503

[27] Noah D Goodman and Andreas Stuhlmüller. 2014. The Design and Implementation of Probabilistic Programming Languages. (2014). Retrieved April 18, 2018 from http://dippl.org

[28] 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 (FOSE 2014)*. ACM, New York, NY, USA, 167–181. https://doi.org/10.1145/2593882.2593900

[29] C. Hopps. 2013. Analysis of an Equal-Cost Multi-Path Algorithm. RFC 2992. (2 March 2013). https://doi.org/10.17487/rfc2992

[30] Daniel Huang, Jean-Baptiste Tristan, and Greg Morrisett. 2017. Compiling Markov Chain Monte Carlo Algorithms for Probabilistic Modeling. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017)*. ACM, New York, NY, USA, 111–125. https://doi.org/10.1145/3062341.3062375

[31] Chung-Kil Hur, Aditya V. Nori, Sriram K. Rajamani, and Selva Samuel. 2014. Slicing Probabilistic Programs. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. ACM, New York, NY, USA, 133–144. https://doi.org/10.1145/2594291.2594303

[32] Teerawat Issariyakul and Ekram Hossain. 2011. *Introduction to network simulator NS2*. Springer Science & Business Media. https://doi.org/10.1007/978-1-4614-1406-3

[33] Nils Jansen, Christian Dehnert, Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Lukas Westhofen. 2016. Bounded Model Checking for Probabilistic Programs. In *Automated Technology for Verification and Analysis*, Cyrille Artho, Axel Legay, and Doron Peled (Eds.). Springer International Publishing, Cham, 68–85. https://doi.org/10.1007/978-3-319-46520-3_5

[34] Srikanth Kandula, Dina Katabi, and Jean-Philippe Vasseur. 2005. Shrink: A Tool for Failure Diagnosis in IP Networks. In *Proceedings of the 2005 ACM SIGCOMM Workshop on Mining Network Data (MineNet '05)*. ACM, New York, NY, USA, 173–178. https://doi.org/10.1145/1080173.1080178

[35] Joost-Pieter Katoen, Annabelle K. McIver, Larissa A. Meinicke, and Carroll C. Morgan. 2010. Linear-invariant Generation for Probabilistic Programs: Automated Support for Proof-based Methods. In *Proceedings of the 17th International Conference on Static Analysis (SAS'10)*. Springer-Verlag, Berlin, Heidelberg, 390–406. https://doi.org/10.1007/978-3-642-15769-1_24

[36] M. Kattenbelt. 2011. *Automated Quantitative Software Verification*. Ph.D. Dissertation. Oxford University, Oxford, UK.

[37] Peyman Kazemian, George Varghese, and Nick McKeown. 2012. Header Space Analysis: Static Checking for Networks. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI'12)*. USENIX Association, Berkeley, CA, USA, 9–9. http://dl.acm.org/citation.cfm?id=2228298.2228311

[38] Ahmed Khurshid, Xuan Zou, Wenxuan Zhou, Matthew Caesar, and P. Brighten Godfrey. 2013. VeriFlow: Verifying Network-wide Invariants in Real Time. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation (NSDI'13)*. USENIX Association, Berkeley, CA, USA, 15–28. http://dl.acm.org/citation.cfm?id=2482626.2482630

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

[40] Dexter Kozen. 2016. Kolmogorov Extension, Martingale Convergence, and Compositionality of Processes. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science, LICS '16, New York, NY, USA, July 5-8, 2016*. 692–699. https://doi.org/10.1145/2933575.2933610

[41] Bob Lantz, Brandon Heller, and Nick McKeown. 2010. A Network in a Laptop: Rapid Prototyping for Software-defined Networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks (Hotnets-IX)*. ACM, New York, NY, USA, Article 19, 6 pages. https://doi.org/10.1145/1868447.1868466

[42] Hyang-Won Lee, Eytan Modiano, and Kayi Lee. 2010. Diverse Routing in Networks with Probabilistic Failures. *IEEE/ACM Trans. Netw.* 18, 6 (Dec. 2010), 1895–1907. https://doi.org/10.1109/TNET.2010.2050490

[43] Vikash Mansinghka, Daniel Selsam, and Yura Perov. 2014. Venture: a Higher-order Probabilistic Programming Platform with Programmable Inference. arXiv:1404.0099 Retrieved from https://arxiv.org/abs/1404.0099.

[44] Piotr Mardziel, Stephen Magill, Michael Hicks, and Mudhakar Srivatsa. 2011. Dynamic Enforcement of Knowledge-Based Security Policies. In *Proceedings of the 2011 IEEE 24th Computer Security Foundations Symposium (CSF '11)*. IEEE Computer Society, Washington, DC, USA, 114–128. https://doi.org/10.1109/CSF.2011.15

[45] David Monniaux. 2000. Abstract Interpretation of Probabilistic Semantics. In *Proceedings of the 7th International Symposium on Static Analysis (SAS '00)*. Springer-Verlag, London, UK, UK, 322–339. https://doi.org/10.1007/978-3-540-45099-3_17

[46] Christopher Monsanto, Joshua Reich, Nate Foster, Jennifer Rexford, and David Walker. 2013. Composing Software-defined Networks. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation (NSDI'13)*. USENIX Association, Berkeley, CA, USA, 1–14. http://dl.acm.org/citation.cfm?id=2482626.2482629

[47] Andrew W. Moore and Denis Zuev. 2005. Internet Traffic Classification Using Bayesian Analysis Techniques. In *Proceedings of the 2005 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '05)*. ACM, New York, NY, USA, 50–60. https://doi.org/10.1145/1064212.1064220

[48] Carroll Morgan, Annabelle McIver, and Karen Seidel. 1996. Probabilistic Predicate Transformers. *ACM Trans. Program. Lang. Syst.* 18, 3 (May 1996), 325–353. https://doi.org/10.1145/229542.229547

[49] J. Moy. 1998. OSPF Version 2. RFC 2328 (Standard). (April 1998). http://www.ietf.org/rfc/rfc2328.txt

[50] Chandrakana Nandi, Dan Grossman, Adrian Sampson, Todd Mytkowicz, and Kathryn S. McKinley. 2017. Debugging Probabilistic Programs. In *Proceedings of the 1st ACM SIGPLAN International Workshop on Machine Learning and Programming Languages (MAPL 2017)*. ACM, New York, NY, USA, 18–26. https://doi.org/10.1145/3088525.3088564

[51] Srinivas Narayana, Mina Tashmasbi Arashloo, Jennifer Rexford, and David Walker. 2016. Compiling Path Queries. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation (NSDI'16)*. USENIX Association, Berkeley, CA, USA, 207–222. http://dl.acm.org/citation.cfm?id=2930611.2930626

[52] Praveen Narayanan, Jacques Carette, Wren Romano, Chung-chieh Shan, and Robert Zinkov. 2016. *Probabilistic Inference by Program Transformation in Hakaru (System Description)*. Springer International Publishing, Cham, 62–79. https://doi.org/10.1007/978-3-319-29604-3_5

[53] Tim Nelson, Andrew D. Ferguson, Michael J. G. Scheer, and Shriram Krishnamurthi. 2014. Tierless Programming and Reasoning for Software-defined Networks. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation (NSDI'14)*. USENIX Association, Berkeley, CA, USA, 519–531. http://dl.acm.org/citation.cfm?id=2616448.2616496

[54] Aditya V. Nori, Chung-Kil Hur, Sriram K. Rajamani, and Selva Samuel. 2014. R2: An Efficient MCMC Sampler for Probabilistic Programs. In *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence (AAAI'14)*. AAAI Press, 2476–2482. http://dl.acm.org/citation.cfm?id=2892753.2892895

[55] Aditya V. Nori, Sherjil Ozair, Sriram K. Rajamani, and Deepak Vijaykeerthy. 2015. Efficient Synthesis of Probabilistic Programs. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15)*. ACM, New York, NY, USA, 208–217. https://doi.org/10.1145/2737924.2737982

[56] ns-3. 2011. Network Simulator. (2011). Retrieved April 18, 2018 from http://www.nsnam.org/

[57] P. Phaal, S. Panchen, and N. McKee. 2001. InMon Corporation's sFlow: A Method for Monitoring Traffic in Switched and Routed Networks. RFC 3176 (Informational). (Sept. 2001). http://www.ietf.org/rfc/rfc3176.txt

[58] Y. Rekhter, T. Li, and S. Hares. 2006. A Border Gateway Protocol 4 (BGP-4). RFC 4271 (Draft Standard). (Jan. 2006). http://www.ietf.org/

rfc/rfc4271.txt

[59] Sriram Sankaranarayanan, Aleksandar Chakarov, and Sumit Gulwani. 2013. Static Analysis for Probabilistic Programs: Inferring Whole Program Properties from Finitely Many Paths. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*. ACM, New York, NY, USA, 447–458. https://doi.org/10.1145/2491956.2462179

[60] Chung-chieh Shan and Norman Ramsey. 2017. Exact Bayesian Inference by Symbolic Disintegration. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*. ACM, New York, NY, USA, 130–144. https://doi.org/10.1145/3009837.3009852

[61] Steffen Smolka, Praveen Kumar, Nate Foster, Dexter Kozen, and Alexandra Silva. 2017. Cantor Meets Scott: Semantic Foundations for Probabilistic Networks. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*. ACM, New York, NY, USA, 557–571. https://doi.org/10.1145/3009837.3009843

[62] Kausik Subramanian, Loris D'Antoni, and Aditya Akella. 2017. Genesis: Synthesizing Forwarding Tables in Multi-tenant Networks. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*. ACM, New York, NY, USA, 572–585. https://doi.org/10.1145/3009837.3009845

[63] Jean-Baptiste Tristan, Daniel Huang, Joseph Tassarotti, Adam Pocock, Stephen J. Green, and Guy L. Steele, Jr. 2014. Augur: Data-parallel Probabilistic Modeling. In *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2 (NIPS'14)*. MIT Press, Cambridge, MA, USA, 2600–2608. http://dl.acm.org/citation.cfm?id=2969033.2969117

[64] Amin Vahdat, Ken Yocum, Kevin Walsh, Priya Mahadevan, Dejan Kostić, Jeff Chase, and David Becker. 2002. Scalability and Accuracy in a Large-scale Network Emulator. In *Proceedings of the 5th Symposium on Operating Systems Design and implementation (OSDI '02)*. USENIX Association, Berkeley, CA, USA, 271–284. http://dl.acm.org/citation.cfm?id=1060289.1060315

[65] András Varga and Rudolf Hornig. 2008. An Overview of the OMNeT++ Simulation Environment. In *Proceedings of the 1st International Conference on Simulation Tools and Techniques for Communications, Networks and Systems & Workshops (Simutools '08)*. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), ICST, Brussels, Belgium, Belgium, Article 60, 10 pages. http://dl.acm.org/citation.cfm?id=1416222.1416290

[66] Nigel Williams, Sebastian Zander, and Grenville Armitage. 2006. A Preliminary Performance Comparison of Five Machine Learning Algorithms for Practical IP Traffic Flow Classification. *SIGCOMM Computer Communication Review* 36, 5 (Oct. 2006), 5–16. https://doi.org/10.1145/1163593.1163596

[67] Frank D. Wood, Jan-Willem van de Meent, and Vikash Mansinghka. 2014. A New Approach to Probabilistic Programming Inference. In *Proceedings of the Seventeenth International Conference on Artificial Intelligence and Statistics, AISTATS 2014, Reykjavik, Iceland, April 22-25, 2014*. 1024–1032. http://jmlr.org/proceedings/papers/v33/wood14.html

[68] Yi Wu, Lei Li, Stuart Russell, and Rastislav Bodik. 2016. Swift: Compiled Inference for Probabilistic Programming Languages. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence (IJCAI'16)*. AAAI Press, 3637–3645. http://dl.acm.org/citation.cfm?id=3061053.3061128