Netter: Probabilistic, Stateful Network Models

Han Zhang*[0000−0001−8740−6502], Chi Zhang, Arthur Azevedo de Amorim[0000−0001−9916−6614], Yuvraj Agarwal, Matt Fredrikson, and Limin Jia[0000−0002−8160−349X]

Carnegie Mellon University, Pittsburgh PA, USA
{hzhang3,yuvraj,mfredrik}@cs.cmu.edu
{chiz5,liminjia}@andrew.cmu.edu, arthur.aa@gmail.com

Abstract. We study the problem of using probabilistic network models to formally analyze their quantitative properties, such as the effect of different load-balancing strategies on the long-term traffic on a server farm. Compared to prior work, we explore a different design space in terms of tradeoffs between model expressiveness and analysis scalability, which we realize in a language we call *Netter*. Netter code is compiled to probabilistic automata, undergoing optimization passes to reduce the state space of the generated models, thus helping verification scale. We evaluate Netter on several case studies, including a probabilistic load balancer, a routing scheme reminiscent of MPLS, and a network defense mechanism against link-flooding attacks. Our results show that Netter can analyze quantitative properties of interesting routing schemes that prior work hadn't addressed, for networks of small size (4–9 nodes and a few different types of flows). Moreover, when specialized to simpler, stateless networks, Netter can parallel the performance of previous state-of-the-art tools, scaling up to millions of nodes.

Keywords: Stateful networks, Probabilistic model checking, Discrete-time Markov chains

1 Introduction

Recent years have seen a surge of interest in automated tools for verifying networks [6,27,36], in particular for analyzing their *quantitative* properties—"What is average latency for this type of traffic?"; "What percentage of packets are dropped on this link?". Such formal verification tools complement other analysis approaches, such as simulations, which are often guaranteed to yield accurate results, but might require a large number of samples to do so.

In contrast to qualitative properties, such as reachability or the absence of routing loops, quantitative properties are often probabilistic, and thus more challenging, due to the complexity of computing over probabilistic models in the presence of the explosion in the number of possible executions. Consider Bayonet [6] for instance, a state-of-the-art language in this domain. Bayonet

* Corresponding author.

can express complex models that account for router state, queue lengths, randomness, and even different packet orderings. Though useful, this expressiveness limits the scalability of the analysis: currently, Bayonet can handle networks of about 30 nodes and small traffic volumes, on the order of 20 packets [6]. Other proposals achieve better scalability by sacrificing expressiveness to varying degrees. McNetKAT [5, 27], for instance, does not model network state or packet interaction, but in return scales to networks with thousands of nodes [27].

The goal of this paper is to seek a different middle ground between expressiveness and scalability. In particular, we aim to analyze the performance of stateful networks *in the long run*, without a priori bounds on the volume of traffic that traverses them. Moreover, we would like to do so while modeling some interaction between different sources of traffic. Potential applications include the analysis of load balancers, traffic engineering schemes, and other components that use states to improve performance. Given the challenges faced by prior work, it is natural to expect that some compromises will have to be made to handle interesting applications. Our hypothesis is that the behavior of many networks should not be too sensitive to the exact ordering of packet arrivals, but rather to how the traffic is distributed among different classes of flows over sizable time intervals—or, put differently, the main interactions between different types of traffic in these networks happen at a large scale. For example, certain traffic engineering schemes (cf. Section 4.2) avoid congestion by periodically reallocating flows on alternative paths based on the volume of data transmitted since the last checkpoint, with typical sampling intervals staying on the order of a few minutes. Based on this insight, we have designed *Netter*, a probabilistic language for modeling and verifying stateful networks. Unlike previous proposals [6, 27], Netter can express interactions between different kinds of traffic while avoiding the combinatorial explosion of having to reason explicitly about all possible packet orderings. Netter programs are compiled to finite-state Markov chains, which can be analyzed by various model checkers, such as PRISM [16] or Storm [8].

We evaluate Netter on a series of case studies: (1) computing failure probabilities on a simple stateless network; (2) a traffic engineering scheme reminiscent of MPLS-TE; (3) a stateful, probabilistic load balancer; and (4) a mitigation strategy for link-flooding attacks from prior work [18]. Our experiments show that Netter can scale to networks of 4–9 nodes, while providing insight into challenging routing questions that prior work had left unaddressed, such as examining the cost of deploying a cheap balancing strategy compared to the optimal one. While these sizes are modest compared to practical networks, we note that Netter can scale to similar orders of magnitude as state-of-the-art tools [27] on the more constrained stateless setting.[1] We expect that Netter's flexibility will allow users to tune between complexity and performance as suits their application.

To summarize, our paper makes the following technical *contributions*:

[1] Due to dependency issues, we only managed to run part of the experiment of Smolka et al. [27], so our comparison is mostly based on the numbers reported by the authors. While this prevents us from making a precise comparison, their setup was similar to ours, and we do not expect the performance of their code to change substantially.

- *Netter*, a domain-specific probabilistic language for modeling and verifying network programs (Sections 2 and 3). By focusing on flow-level modeling and abstracting away from event orderings, Netter can verify asymptotic properties of stateful networks that were previously out of reach [6, 27].
- Optimizations for reducing the size of the automata generated by Netter (Section 3). To be confident in their correctness, these optimizations were verified in the Coq proof assistant [30].
- A series of case studies evaluating the expressiveness and scalability of Netter (Section 4), including a comparative benchmark, a traffic engineering scheme, a stateful load balancer, and a defense against link-flooding attacks. Our results show that Netter scales similarly to state-of-the-art tools on stateless networks, while enabling the automated quantitative analysis of some stateful routing schemes that prior work could not handle.
- Netter is open-source, and available at https://github.com/arthuraa/netter. Our artifacts including case study code and Coq formalization are public available [39] and can be used with the VMCAI virtual machine [7].

The rest of the paper is organized as follows. In Section 2, we give a brief overview of the Netter workflow with a simple, but detailed, model of a stateful network, showing how we can reason about its performance characteristics automatically. In Section 3, we delve into the Netter language, covering its main functionality and how it departs from prior work. We describe its Haskell implementation and the optimization passes used to improve model checking time. In Section 4, we present our case studies. Finally, we discuss related work in Section 5 and conclude in Section 6.

2 Overview

The workflow of using Netter for analyzing probabilistic quantitative network properties is depicted in Figure 1. Users provide a Haskell file with a model written in an embedded probabilistic imperative language. The model contains the code of the network, as well as declarations of key performance metrics that the users want to analyze, such as the drop rate of the network. The Netter compiler transforms this model into a probabilistic automaton that encodes a finite-state, discrete-time Markov chain. This automaton is encoded in the language of the PRISM model checker [16], but back-ends to similar tools could be added easily. This Markov chain is then fed to Storm [8], a high-performance probabilistic model checker, along with a set of properties to analyze, such as the expected value of a performance metric in the stationary distribution of the chain.

To illustrate this workflow, consider the network depicted in Figure 2. Two servers, S1 and S2, sit behind a load balancer LB. When a new flow of traffic arrives, the load balancer simply picks one of the servers at random and routes the traffic through the corresponding link. Any traffic that exceeds the link's capacity is dropped, and we are interested in computing the long-term average drop rate of this load-balancing scheme under certain traffic assumptions.

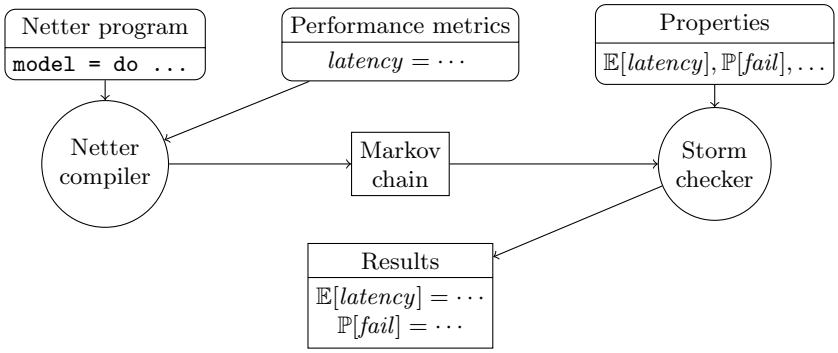


Fig. 1. Netter workflow. Rounded corners denote user-provided inputs, rectangles denote outputs, and circles denote components.

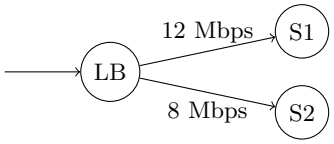


Fig. 2. Example network with two servers, S1 and S2, and a load balancer LB. LB has randomly forward incoming flows to either server. If the new flow traffic size is 10 Mbps, it will experience losses if forwarded to S2.

Figure 3 shows a model of this network in Netter. Unlike other network modeling languages [6], Netter does not represent different network nodes as separate entities. Rather, the model presents a global view of the network, where all the state is manipulated by a single program. In our example, this state comprises just the two allocated variables `dest` and `newFlow` (lines 10–11), which represent the state of the load balancer and the state of the current flow. Later examples will show models where other nodes also use state (Section 4).

Since Netter programs are compiled to finite-state automata, variables must be bounded, and their bounds are specified in the parameters of the `var` function. The `dest` variable represents the router chosen by the load balancer, while `newFlow` tracks whether a new flow of traffic has just arrived. For now, we assume that there can be only one flow at a time in this network, which transmits data at a rate of 10 Mbps. The `rewards` call (lines 13–15) specifies that we are interested in analyzing the average drop rate of the network. Note that to avoid confusion with standard Haskell functions, many Netter operators are primed (e.g. `when'`) or prefixed with a dot (e.g. `./`). The `.? ... .:` form is the Netter equivalent of the ternary operator in C-like languages.

Figure 3 shows the code representing the execution of one step of the Markov chain. On the first time step, the state variables are set to their lower bound. Then, the program enters a loop, executing its code once per step. More precisely,

```
1  trafficSize = 10
2  bandwidth1 = 12
3  bandwidth2 = 8
4
5  dropRate bw = int dropped ./ int trafficSize
6    where dropped = max 0 (trafficSize - bw)
7
8  model :: Prog ()
9  model = do
10   dest <- var 1 2
11   newFlow <- var 0 1
12
13   rewards "dropRate"
14     (dest .== 1 .? dropRate bandwidth1
15      .: dropRate bandwidth2)
16
17   newFlow .<-$ [(0.1, 1), (0.9, 0)]
18   when' (newFlow .== 1) $ do
19     dest .<-$ [(0.5, 1), (0.5, 2)]
```

Fig. 3. Netter implementation of model in Figure 2.

the behavior of the network is defined in lines 17–19. Line 17 says that on every step there is a 10% probability that a flow ends and a new one starts. The `.<-$` operator samples from a probability distribution specified as a list of probability/value pairs. By changing this distribution, or by adding more state variables, we can model richer traffic patterns (Section 4.3). Lines 18–19 say that, whenever a new flow appears, the load balancer chooses its server uniformly at random. The selected server remains stored in `dest` for the next few time steps, until `newFlow` becomes 1 again.

Figure 4 shows the probabilistic automaton produced by Netter for this network. Though the execution of the original program happens conceptually in one step, representing this directly as a probabilistic automaton is challenging. It is easier to decompose the code into a series of more elementary steps, where we use the auxiliary *program counter* variable `pc_0` to choose which command of the program needs to be executed next. The point `pc_0 = 0` marks the beginning of the execution at the source level. (Using a small number of PC values is crucial for keeping the generated model small, an issue that prior work on Probabilistic NetKAT also faced [27]; cf. Section 3.) We set the `dropRate` reward to zero at other PCs to ensure that it is counted only once per source-level step.

To calculate the long-term average drop rate for the incoming flow, we feed Storm the PCTL query `R{"dropRate"}=? [LRA] / LRA=? [pc_0=0]`, to which it replies `0.1`. (The adjusting factor `LRA=? [pc_0=0]` is included to compensate for intermediate steps in the automaton that have no source-level counterpart.)

```
1  module m0  // load-balance new flow to server
2  dest_0 : [1..2];
3  [step] (pc_0!=1) -> true;
4  [step] ((pc_0=1)&(newFlow_0=1)) -> 0.5:(dest_0'=1) \
5                                     + 0.5:(dest_0'=2);
6  [step] ((pc_0=1)&(newFlow_0!=1)) -> (dest_0'=dest_0);
7  endmodule
8
9  module m1  // randomly decide if flow becomes active
10 newFlow_0 : [0..1];
11 [step] (pc_0!=2) -> true;
12 [step] (pc_0=2) -> 0.1:(newFlow_0'=1) \
13                    + 0.9:(newFlow_0'=0);
14 endmodule
15
16 module m2  // manage pc counter
17 pc_0 : [0..2];
18 [step] (pc_0=0) -> (pc_0'=2);
19 [step] (pc_0=1) -> (pc_0'=0);
20 [step] (pc_0=2) -> (pc_0'=1);
21 endmodule
22
23 rewards "dropRate"
24 (pc_0=0) : ((dest_0=1)?0.0:0.2);
25 (pc_0!=0) : 0;
26 endrewards
```

Fig. 4. Compiled DTMC model from Netter implementation in Figure 3.

3 The Netter Language

Operations Figure 5 enumerates some of the main operations in Netter. The `Expr` type represents integer and boolean values in the network program. The type `Prog` is used for commands and program declarations. It carries the structure of a monad [22], allowing us to easily compose subprograms. Commands have a return type of `()`, the unit type, which means that they yield no values, and are run solely for building the program. Other declarations, however, may produce useful results, such as the variable declaration command `var`, which returns an `Expr` (cf. Figure 3).

The API highlights important distinctions with respect to prior work. First, unlike Probabilistic NetKAT [5] or Bayonet [6], there are no specialized commands for manipulating packets (though this functionality can be encoded with regular state variables, as we do in Section 4.1). Indeed, since Netter is tailored to analyze flow-level behavior, we focus on commands that manipulate the high-level routing decisions (e.g. the `dest` variable in Figure 3), and assume that these can be implemented in terms of lower-level packet-manipulating primitives. Sec-

Function	Type	Description
`var`	`Int -> Int -> Prog Expr`	Declare a state variable
`.+, .-, .&&`	`Expr -> Expr -> Expr`	Arithmetic and logic
`.<-`	`Expr -> Expr -> Prog ()`	Deterministic assignment
`.<-$`	`Expr -> [(Double, Expr)] -> Prog ()`	Probabilistic assignment
`if'`	`Expr -> Prog () -> Prog () -> Prog ()`	Conditional
`block`	`Prog () -> Prog ()`	Block for local variables
`.!!`	`[Expr] -> Expr -> Expr`	List indexing

Fig. 5. Select Netter primitives.

```
data RouterState = RouterState { reservedBandwidth :: Expr
                               , sampledBandwidth :: Expr }

makeRouter :: Prog RouterState
makeRouter = do
  rb <- var 0 maxBandwidth
  sb <- var 0 maxBandwidth
  return (RouterState { reservedBandwidth = rb, sampledBandwidth = sb })
```

Fig. 6. Netter programs can use Haskell abstractions such as functions and data types.

ond, unlike Probabilistic NetKAT, Netter programs can use *arbitrary* expressions in assignments and in the guards of if statements, making it easy to encode typical imperative programs—indeed, most of the case studies in Section 4 use this functionality.[2] On the other hand, Probabilistic NetKAT allows programs to perform unbounded iteration, while Netter does not have an analogous construct. This simplifies the semantics of Netter programs, which can be easily described as a stochastic matrix on the finite space of all program states. An entry M_{ij} of this matrix describes the probability of transitioning from state i to state j after running the code. The semantics is similar to that of McNetKAT [27], but includes a semantics for arithmetic expressions, and omits a clause for iteration. (In practice, we have not found the absence of loops to be a limitation, since we could partly emulate iteration by wrapping Netter commands in Haskell loops.)

Netter has a phase distinction between *model code*, which is analyzed by the model checker, and *compiler code*, which is responsible for generating the former. The `Expr` type produces expressions that are consumed by the model checker, and thus does not have a well-defined "value" when the model is being generated. This prevents us from operating on expressions as if they were regular values in Haskell; for instance, we cannot test if two model variables hold the same value by writing `x == y`, since the equality operator returns a fixed boolean rather than a symbolic expression. This is why many basic Haskell operators have counterparts for Netter expressions, as we have seen with the `./` and `.==`

[2] In principle, since variables are bounded, it would be possible to do away with expressions by evaluating them at every possible state. However, this would result in much larger compiled models, making the analysis more costly.

operators of Figure 3. Despite this phase distinction, the compiler code is free to use other Haskell types and operations to generate a model, which makes up for the minimalist set of basic constructs available in Netter. For instance, we can represent a stateful router with a record that contains Netter variables (cf. Figure 6), a functionality that is useful when defining complex network models, as we will see in Section 4.

Implementation The compilation process that takes a network algorithm implemented in Netter and generates an *optimized* PRISM model was implemented in about 2k lines of Haskell. First, user-level commands are processed to build an internal representation of a model in a simple imperative language called Imp. As depicted in Figure 7, the program then undergoes a series of compilation passes to produce a Markov-chain model. An important part of this process is the translation of the program to a control-flow graph (CFG), which can be more directly represented as an automaton. The size of the resulting automaton is linear in the size of the CFG, which must be kept to a minimum to avoid blowing up the state space. This is the job of two optimization passes of the pipeline: one that inlines as many assignments as possible, and one that removes stores and variables that are not used to compute the rewards declared by the user. The inlining pass is particularly challenging. Indeed, in an earlier version of the compiler, we tried to symbolically execute the Netter program to remove the need for any intermediate assignments, probabilistic and deterministic alike. However, composing multiple symbolic probabilistic assignments can quickly lead to bloated models: if an assignment with n probabilistic branches is expanded in another probabilistic assignment with m branches the result is generally a probabilistic assignment with $n \cdot m$ branches.

To avoid this issue, we adopted a more conservative strategy where we only inline deterministic assignments. This requires some care: if a variable x receives the result of a random sample, we need to stop propagating any inlined expressions that mention x, since they refer to its old value. To increase our confidence in this step, we have formalized our main optimization passes using the Coq proof assistant [30], and manually translated the algorithms to Haskell. To define the semantics of the language, we formalized a core of finite probability theory in Coq, including infrastructure for reasoning about coupling arguments [10]. Probabilistic NetKAT relies on a similar optimization to compile to PRISM [27], though its logic is considerably simpler, since only constants can be assigned in the language (and thus almost no dependencies need to be considered).

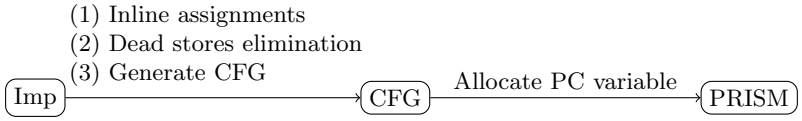


Fig. 7. Netter compilation pipeline.

4 Applications

Now that we have acquainted ourselves with the Netter basics, we discuss four case studies that used the language to model and analyze stateful networks. For all of the following cases, we evaluate Netter on a 12-core VM with 64 GB memory. Unless specified otherwise, we use the Storm [8] model checker with their Sparse backend engine. We set the max memory limit to be 40 GB for Storm, reserving headroom for graceful shutdown in case of memory exhaustion.

4.1 Warming Up

As a first case study, we evaluate the performance of Netter on a simple stateless benchmark. This benchmark exercises features that could already be handled in prior work [6,27], and is a sanity check to ensure that Netter's expressiveness does not incur large performance penalties when it is not needed. Figure 8 presents a network that connects two hosts, H_1 and H_2, via a chain of $4k$ intermediate switches. Each switch $S_{i,1}$ forwards traffic to either $S_{i,2}$ or $S_{i,3}$ with equal probability. Both $S_{i,2}$ and $S_{i,3}$ forward to $S_{i,4}$, but the link $S_{i,3} \to S_{i,4}$ can fail with probability $p = 10^{-3}$. Finally, $S_{i,4}$ forwards to $S_{i+1,1}$. We are interested in the probability that a packet is successfully delivered from H_1 to H_2.

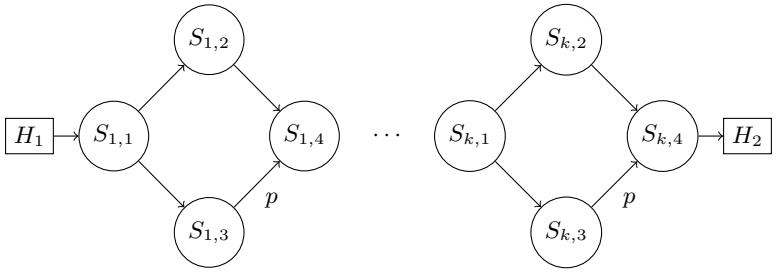


Fig. 8. Chain topology with failures.

We compare the time taken to check the Netter model in Storm against the time taken by PRISM and Storm to check a handwritten model of this network. The handwritten model was taken from an analogous experiment in the McNetKAT paper [27]. We set a timeout of 5 minutes. Figure 9 presents the results. We observe that Storm takes about the same time to check the Netter and the handwritten models, with the latter being slightly cheaper to process. We checked networks of at most 4M switches. PRISM timed out at 32k switches. For comparison, the authors of McNetKAT [27] reported that their custom solver could analyze 65k switches in 2.5 minutes running on a cluster with 24 machines, while Bayonet could analyze 32 switches in 25 minutes. Moreover, their performance figures for checking the handwritten model with PRISM are

similar to ours. We did not manage to run the McNetKAT code in our setting, due to dependency issues.

The gap between Bayonet and the other tools is to be expected, as it is based on a much more general solver and accounts for traffic details that the others don't, such as asynchronous event scheduling. (Note that these features should not fundamentally change the analysis, since this experiment considers only one packet.) As for McNetKAT, we believe that the difference in performance can be mostly explained by the use of the Storm back-end; nevertheless, since Storm is compatible with the PRISM language, which can also be targeted by McNetKAT, McNetKAT could readily benefit from advances in other model checkers as well.

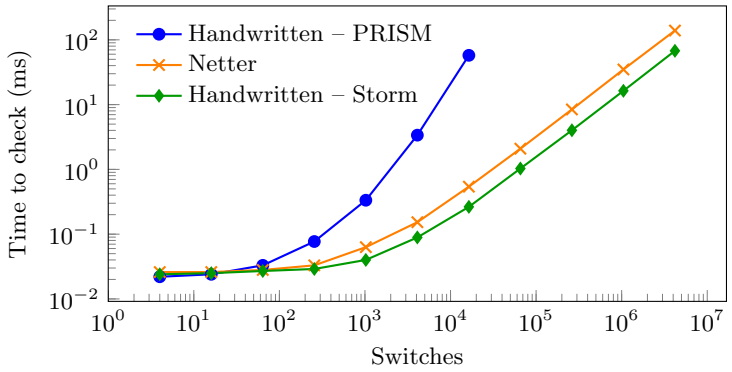


Fig. 9. Checking performance of chain example.

The case for using Netter on an example like this is not so strong, as the handwritten automaton for PRISM is about the same size as our model, and can be checked slightly faster. With the next case studies, we will see examples with non-trivial control flow that would be difficult to encode directly in PRISM.

4.2 Traffic Engineering with MPLS

Traditional IP routing can be too inflexible for traffic-engineering purposes, as every packet is sent through the shortest path between its source and destination. If a link on this path becomes congested, the performance of the network degrades. In modern networks, a popular solution is to manage sections of a path using Multiprotocol Label Switching (MPLS) instead of pure IP.

Originally, MPLS was introduced to speed up the handling of addresses in switches. When packets from other protocols enter an MPLS network, they are encapsulated with a *label* determined by their header and routed through a path in the network until they reach an exit node, when the labels are removed. (In reality, the labels change as packets traverse the network, but this detail is not relevant for our purposes.) Labels are processed in a way that resembles IP

addresses, in that the next hop of a packet along a path is determined by its labels. However, labels are much shorter than IP addresses, and thus faster to process in hardware.

As MPLS was extended over the years, it gained the ability to manage labels with more flexibility than IP, making it attractive for traffic engineering. To avoid congestion, for example, MPLS can reserve some bandwidth for each label, and assign them to paths so that the total reserved bandwidth on each link does not exceed its capacity. Moreover, this reserved bandwidth can change dynamically based on traffic demands, causing labels to be reallocated on different paths.

Prior work [24] observed that bad MPLS configurations can allocate labels on sub-optimal paths, leading to *latency inflation*. In the network analyzed by the authors, the weighted latency was 10%–22% higher than the optimal, and some labels could remain on sub-optimal paths for as long as 10 days!

To investigate the causes of latency inflation, we devised an experiment that models the bandwidth adjustment logic used by the main network vendors—the so-called "auto-bandwidth" feature. In this experiment, each flow corresponds to the traffic assigned to one MPLS label; thus, each flow is routed through a particular path in the network, and has a certain bandwidth reserved for it along this path. Our adjustment logic is governed by two parameters: the *sample interval* and the *adjustment interval*. Every sample interval, the network samples the volume of traffic on each flow. When the adjustment interval is completed, the largest sample since the beginning of the interval is compared against the current reserved bandwidth for each flow. If the sample is larger than the reserved bandwidth, the network reallocates the flow on a new path with enough bandwidth, potentially evicting lower-priority flows allocated there. We set the sample interval to 3 time steps, and the adjustment interval to 9 time steps. (Real MPLS deployments expose other configuration options as well, such as the adjustment threshold. For simplicity, we omit those.)

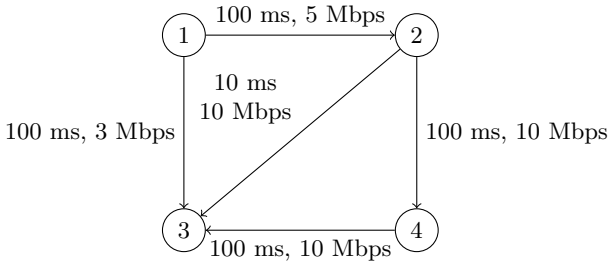


Fig. 10. MPLS network topology.

Figure 10 shows the topology used in our experiment. The link between 2 and 3 is a local link with high bandwidth, whereas the other four are long links with higher latency. There are two flows in this network: $f_{2,3}$, a high-volume flow of 9 Mbps between 2 and 3, and $f_{1,3}$ between 1 and 3. The volume of traffic

in $f_{1,3}$ varies between 2 and 4 Mbps according to a random walk, moving up or down if possible with a probability of 25%.

We use Storm to compare the long-term weighted latency of two configurations for this network: one where $f_{2,3}$ has a higher priority than $f_{1,3}$, and the other one where the priorities are reversed. We switch to the Hybrid engine to avoid memory exhaustion for this model. Storm reports that the weighted latency is 101ms for the first configuration 81ms for the second one, which corresponds to an increase of about 24%. Intuitively, in the first configuration, when $f_{1,3}$ triggers an adjustment, it ends up evicting $f_{2,3}$ to a much longer path, because it has higher priority. Since $f_{2,3}$ carries more traffic, the weighted latency goes up. In the second configuration, instead, $f_{1,3}$ is reassigned to the path $1 \to 2 \to 4 \to 3$, and the problem does not arise. These observations corroborate the aforementioned empirical results.

As a side note, this case study was one of the original motivations for implementing Netter. We attempted to encode an earlier version of this model directly in the PRISM language, but felt that correctly expressing the control-flow of the autobandwidth logic as an automaton was error prone, especially when trying to express more complex topologies and flow configurations, since there is no convenient way of abstracting the network topology in PRISM. By contrast, our Netter model takes in a high-level description of the network topology as an adjacency matrix, and automatically computes the lists of possible routes for a flow ranked by latency, while ensuring that the available bandwidth on each route is correctly updated.

4.3 Stateful Load Balancers

Load balancers are commonly used to improve web application performance by sharing and distributing a pool of resources. They act as virtual servers to receive incoming client requests and forward requests across different backend servers to manage desirable loads between servers. Many load balancing algorithms require storing internal state information. For example, a Round Robin algorithm needs to remember which server it assigns the previous flows before allocating the next one. Many other algorithms need to compare the current server loads before finding a suitable candidate to allocate the new flow. Although previous works have modeled randomized load balancers [5, 6], they do not support the case for stateful load balancers or other complex algorithms. In our experiment, we implement and analyze three different load balancing algorithms.

Max Free Capacity. The Max Free Capacity algorithm requires the load balancer to forward new flows to the server with the largest available capacity, measured as the difference between the server's maximum capacity and the its current load. Intuitively, this strategy should have a very low probability of flow loss because it can find the best server to process all incoming flows. However, the disadvantage is that the load balancer needs to collect information from all servers before making any decisions. This method can be costly, generating too much internal traffic and incurring high latency.

Round Robin. The Round Robin algorithm uses an internal counter to assign new flows to servers. When a new flow arrives, the load balancer forwards it to the server given by the current counter and updates the counter to the next server in line. The advantage of this strategy is that the load balancer doesn't need to check the server load at all. The disadvantage is that it may cause significant server imbalances and packet drops when a server receives too much load while another one is mostly free.

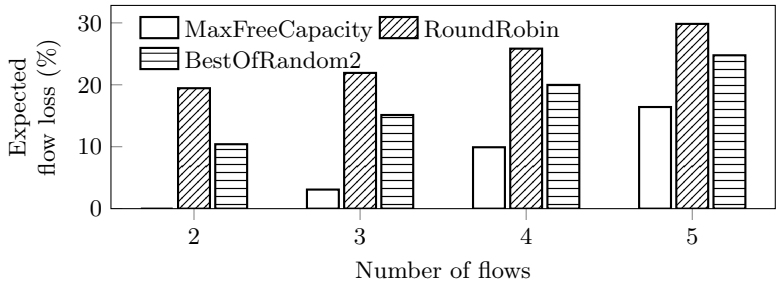
Best of Random 2. In the Best of Random 2 algorithm, the load balancer randomly picks two servers, compares their current load, and sends the new flow to server with the largest available capacity. This simple but powerful algorithm is proven to have a small maximum server load with high probability [21], making it a popular option for many applications [29]. Compared to the previous two algorithms, the Best of Random 2 algorithm is a compromise for avoiding flow losses while reducing internal traffic queries, since it only needs to query two servers instead of all of them.

We implement these three algorithms in Netter in a simple load balancing use case—we put one load balancer in front of a group of servers, and the load balancer forwards incoming flows to any one of the servers. We explore various settings in terms of the number of servers, number of flows, and server capacity, and compare these algorithms' performance. We pick two metrics—average flow loss rate and server load imbalance—to measure these algorithms' performance. Finally, we show the complexity of running Netter in these cases.

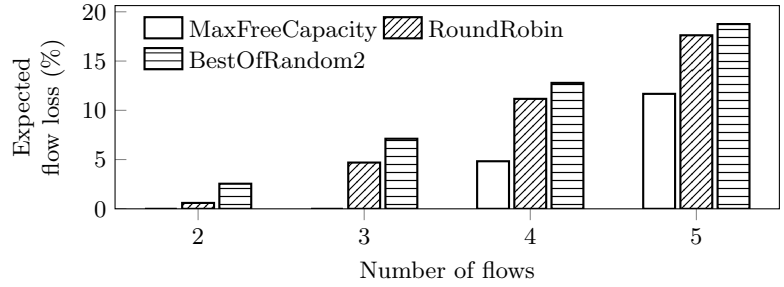
During our evaluation, we represent flow traffic with a Markov model. At every timestamp, each inactive flow can be independently activated with a probability of 0.6. Upon activation, the flow randomly selects a volume between 1 and 3 Mbps. Active flows also have a 0.6 probability of deactivating and return to the inactive state at every timestamp. In the following graph, the flow number represents the maximum possible number of flows that could arrive at the load balancer at the same timestamp.

Flow Loss Rates. Figure 11a shows the long-run flow loss percentage for different algorithms in a group of 3 servers with *different capacity*. The percentage of flow loss is calculated by the number of packet drops over total incoming loads. From the figure, we can see that the Max Free Capacity algorithm always has a strictly lower flow loss compared to the other two algorithms. The Round Robin algorithm causes more flow losses than the Best of Random 2 algorithm. This is because the Best of Random 2 will overload a server beyond its capacity only if neither of the two randomly chosen servers has enough free capacity. In comparison, the Round Robin algorithm is agnostic to the server's current load, and it assigns flows periodically in the long run. We can also see that the difference in flow loss for these three strategies is getting smaller when the flow load increases. This is because none of the strategies deal with the case where the incoming flow load is larger than the total server capacity.

This result confirms our intuition that the Max Free Capacity is the best solution among the three algorithms in reliable flow allocation if we ignore its



(a) Different capacity, number of servers=3.



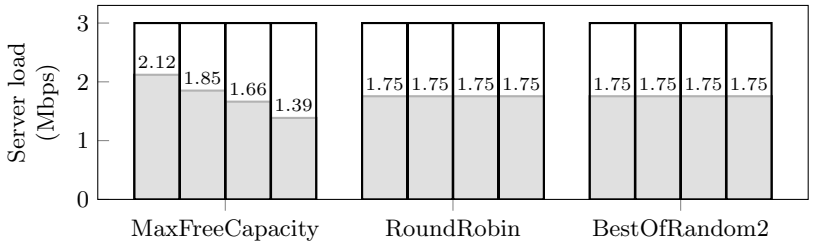
(b) Equal capacity, number of servers=3.

Fig. 11. Model checking results on flow loss rates for the load balancer using different algorithms to allocate flows to 3 servers.

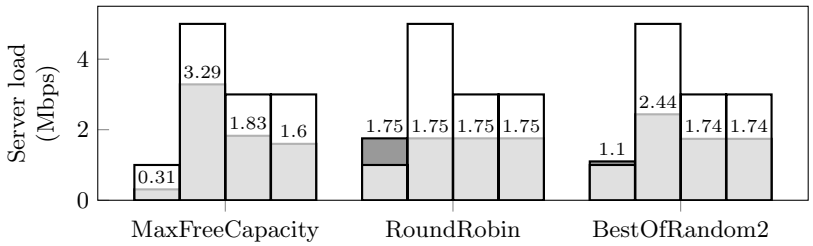
costly nature of querying every server for their load during runtime. Moreover, we can see that the Round Robin algorithm has a larger flow loss rate than the other two algorithms when the servers have different capacity.

Figure 11b shows the long-run flow loss percentage for every algorithm with servers of the *same capacity*. Comparing to the previous figure, we can see the difference in the flow loss rate across all three algorithms. Each algorithm observes less flow loss comparing to the previous case. Similarly, the Max Free Capacity algorithm has the lowest flow loss rate. An interesting observation for this case is that the Round Robin algorithm causes less flow loss than the Best of Random 2 algorithm. This is because, given the parameters set for the Markov flow model, the server is less likely to have an active flow when selected by the Round Robin algorithm (i.e., previous flows must remain active for many rounds) than with the Best of Random 2 algorithm (i.e., server can be picked at any time).

Server Load Imbalances. Figure 12a shows an imbalance among servers by plotting the absolute value of servers' capacity and load. In this case, all servers with the same 3 Mbps capacity are represented as white bars, and the grey bars show the long-run load on the servers. We can see that the Round Robin algorithm and the Best of Random algorithm have the same load allocation among the servers. Yet, the Max Free Capacity algorithm's imbalance is caused by a prior-



(a) Equal capacity (3 Mbps each).



(b) Differnet capacity (1, 5, 3, 3 Mbps).

Fig. 12. Model checking results on server load imbalance from the three load balancing algorithms to allocate 5 flows to 4 servers. For each algorithm, the four bars represents the four individual servers. White bar indicates server capacity, light grey for average loads, and dark grey for overloads.

ity among servers when breaking a tie. By default, this algorithm allocates the flow to the server with a smaller index when servers have the same free capacity.

Figure 12b shows the imbalances between servers with unequal capacity. We assign each server with 1 Mbps, 5 Mbps, 3 Mbps, and 3 Mbps maximum capacity, respectively. The dark grey bars in the figure represent overloading servers beyond their maximum capacity. For example, the Round Robin algorithm allocates 1.78 Mbps traffic to server 1, where over 40% will be dropped due to server overload. The root cause of the massive server overload is that the Round Robin algorithm is agnostic to individual server loads when allocating new flows. For simplicity, the load balancer only keeps one next counter to decide where to send the next flow. As a result, the load balancer forwards the incoming flow to the chosen destination, even if the server is busy while others have ample free capacity. In comparison, the Best of Random 2 significantly reduces server utilization imbalance. Instead of deterministically assigning one server for new flows, the load balancer randomly picks two candidates and checks their utilization. Server overload can still happen—when the load balancer picks two busy servers by chance, and neither one can fulfill the new flow without loss. However, the probability of server overload (and imbalance flow allocation) is much smaller than the Round Robin algorithm. On the other hand, the Max Free Capacity algorithm is the optimal strategy in avoiding server overload. This is because

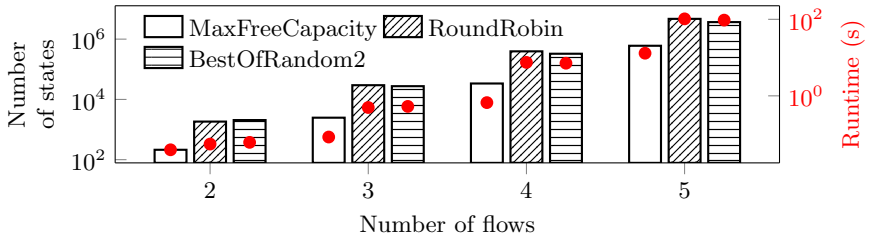


Fig. 13. Model checking complexity in Markov chain state number and runtime for load balancer using each of the three algorithm to allocate flows among 4 servers with same capacity. The runtime is marked as red point scaled on the right side.

the load balancer has a global view of server loads before allocating new flows. It can always pick the capable server to process the new flow if such one exists. Therefore, we observe that no server experiences high or near-max loads.

Analysis Runtime. Figure 13 shows the complexity and runtime of using Netter to analyze different algorithms. In the figure, we show that the runtime of model checking is proportional to the number of states the Markov chain needs to calculate. Since different algorithms have different numbers of variables in the model, they vary significantly in model complexity. The Best of Random 2 algorithm uses random choice in picking the servers, which leads to more states for the model checker to explore, especially when comparing against the Max Free Capacity algorithm. The Round Robin algorithm requires a global variable in the model to keep track of the next server to allocate new flow. In addition to the complexity in the algorithm, we must keep auxiliary variables such as flow assignment and flow volume to calculate flow loss rate and server loads at every timestamp. These variables increase the state space exponentially, and thus pose a scalability challenge for exhaustive model checking.

We empirically evaluate different model-checking engines of Storm. We compared their Sparse and Hybrid engines and found out the Sparse engine is universally faster in solving the load balancer models. The Hybrid engine, on the other hand, sacrifices runtime speed for smaller memory usages. Therefore, for problems with larger state space that Sparse engine runs out of memory, we use the hybrid engine. One example is the case study of MPLS in Section 4.2.

4.4 Defending against Link-Flooding Attacks

Link-flooding attacks have become a serious threat to Internet security [34]. As a distributed denial-of-service attack, link-flooding aims to disrupt the availability of specific links between routers (e.g., data centers, Internet exchange points, Autonomous Systems). All services and connections sharing the same victim links in their paths will be affected, regardless of their sources and destinations.

To defend against link-flooding attacks, several prior methods propose routing-based mechanisms to divert the victim's network traffic during attacks [18, 25].

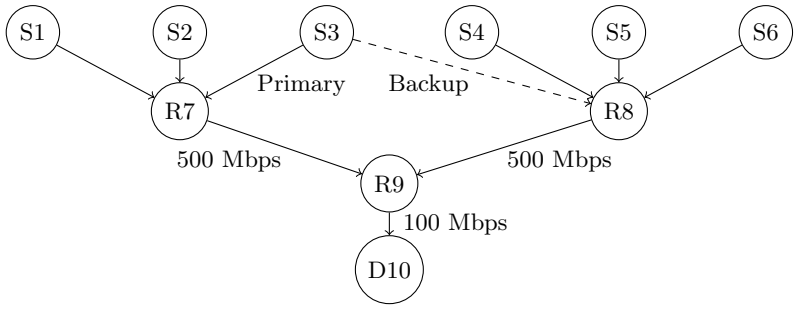


Fig. 14. Topology for DDoS case study in Section 4.4. S3 uses R7 as its primary link, but maintains an alternative path through R8 as a backup.

They propose different rate-limiting and routing algorithms to be applied at intermediate routers to coordinate defenses. When hosts detect links in their routing path are under attack, they proactively switch to other alternative routing paths to avoid such links. Although such approaches are intuitive, many practical challenges affect the feasibility of re-routing to alternative paths [31]. For example, before switching, the victim host needs to analyze what latency and bandwidth availability the secondary path can provide. Therefore, it is important to verify the effectiveness and applicability of re-routing techniques based on the specific routing algorithms and global topology.

To illustrate, we present a network topology vulnerable to such attacks (cf. Figure 14). This topology is adapted from the evaluation in the CoDef paper [18], where they implement a simulation network to measure the effectiveness of CoDef rate-limiting and routing algorithm. Node S1-S6 are clients sending traffic to D10 as the final destination, R7-R9 are intermediate routers. S1 and S2 are malicious attackers sending a massive amount of traffic. Both of them send a median of 300 Mbps of traffic following a Pareto distribution. S3 is the victim of the example. It shares the same link as attackers S1 and S2. However, it can switch to its backup link to avoid attackers. S3 and S4 are file transfer applications, greedily expecting to utilize as much bandwidth as possible. S5 and S6 consume 10 Mbps consistently, representing fixed bandwidth flows such as streaming. Meanwhile, S3 maintains a secondary backup link with router R8, but it does not utilize that link under normal circumstances. Using multiple routers from different providers, as S3 does, is commonly known as *multihoming* [1].

We implement three routing algorithms in Netter using the same topology and check the average bandwidth allocated for each flow as properties. To switch between different algorithms, we only need to change the code in the routing module, keeping the rest of the model the same.

CoDef Collaborative Routing. CoDef proposes a collaborative routing algorithm for routers R7-R9 to coordinate and defend against link-flooding attackers S1 and S2. All routers collaboratively label each destination-source pair as a unique

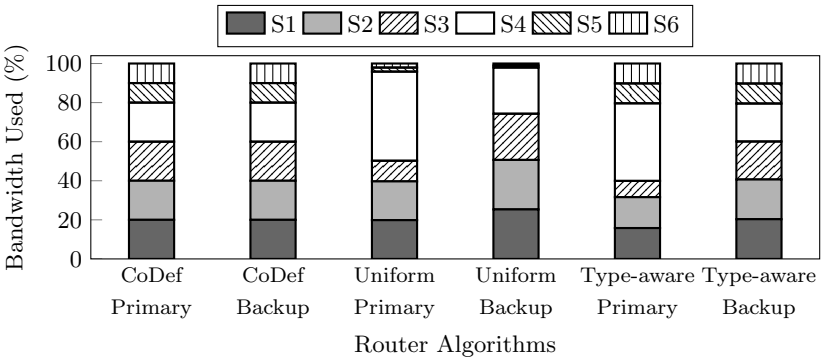


Fig. 15. Model checking results for each flow's portion at the destination (Node 10). S1 and S2 are attacker flows. Each bar indicates a corresponding router algorithm and which router link flow 3 connects to.

path and ensure fair bandwidth utilization on a per-path basis. The algorithm allocates the bandwidth in two passes: first, the router assigns a fair share to each flow, and then allocate additional free bandwidth in a second pass.

Uniformly Random. This algorithm allocates egress bandwidth uniform-randomly based on the proportion of incoming bandwidth. It is a statistically simple allocation algorithm and requires minimal metadata communication between routers. For example, suppose S1 and S2 send a large amount of traffic to router R7 and S3 uses its primary link. The router will allocate a higher share of the available 500 Mbps outgoing bandwidth to these two flows, reducing flow 3's portion.

Type-aware Priority. The type-aware routing algorithm considers the type of flows. Motivated by the real-world example of classifying traffic into several classes and provide different quality-of-service guarantees [3, 24], routers can assign priorities and available bandwidth accordingly to a different type of traffic. In our example, we enable the routers to prioritize consistent flow transmission; specifically, the fixed bitrate flows 5 and 6. As for the remaining flows, the router follows the same uniform strategy to allocate the available bandwidth.

Figure 15 compares the model checking results for different algorithms. CoDef algorithm enforces fair allocations regardless of S3's egress link. However, other algorithms present pathological cases when S3 is under direct link-flooding attacks from attackers S1 and S2. The Uniform algorithm equally splits bandwidth between two incoming routers ($7 \to 9$, $8 \to 9$). In the Primary case, flows 1-3 and 4-6 use an equal amount, while in Secondary flows 1-2 and 3-6 are the same. The Type-aware algorithm reserves a high priority to flow 5 and 6, and uniformly shares the rest bandwidth among flows 1-4.

We successfully used Netter to analyze a large topology of 10 nodes without the need to scale all numbers by a common factor manually. Switching between the router's rate-limiting algorithms is also relatively easy. We can implement a

new rate-limiting algorithm and specify which router to use it. One limitation of Netter model checking is that we cannot use continuous distributions for traffic volumes. Because Netter compiles programs to finite-state models, we use a discrete approximation of the Pareto distribution to represent attacker S1 and S2's traffic model. We calculate the numerical value for every 5th percentile along with their cumulative density function and use these numbers and their percentiles to approximate the probabilistic distribution of flows 1 and 2.

5 Related Work

There is a rich body of work for testing and verifying forwarding behaviors such as reachability and loop freedom, in stateless networks [9,13–15,19,20,32,33,35, 37,38] and stateful networks [4,23,28,36]. The aforementioned projects do not concern quantitative properties of the network such as latency and throughput, which our work focuses on. Moreover, we support probabilistic network models, which differentiates our work from quantitative network analysis based on fixed quantity and SAT solvers [11,12,17].

Next, we discuss related work that is closer to ours, on probabilistic languages to model and analyze quantitative properties of networks [6,27]. NetKAT [2,5, 26,27] is a family of network-modeling languages based on Kleene algebra with tests. In the original NetKAT [2], a program denotes a set of packet histories, which are traces of the states of a packet while it traverses the network. Probabilistic NetKAT [5,26,27] adds in probabilistic choice, and has been used to analyze interesting case studies, such as fault tolerance of a data center design [27]. The semantics of Probabilistic NetKAT is similar to Netter's, though the more sophisticated features of the language go beyond finite-state Markov chains, and require continuous distributions. Language-wise, Netter and Probabilistic NetKAT are built on similar primitives, with two main differences: (1) NetKAT programs can only assign constants to variables, whereas Netter programs can assign arbitrary arithmetic and logical expressions, and (2) NetKAT has unbounded iteration, which Netter does not (though Netter programs can simulate iteration by metaprogramming; that is, by writing a Haskell loop that repeatedly calls a code snippet). The restricted assignments are not too limiting for NetKAT, since it is used to model stateless networks, and the assignments encode a network node's actions after matching on the headers of a packet ("if the destination IP is 10.0.0.1, forward the packet to port 10"). Netter, on the other hand, was primarily designed to model stateful networks, which perform more assignments and would be awkward to encode without expressions. Because of its richer assignments, Netter's optimizations are more challenging than NetKAT's [27], since they involve inlining expressions, tracking state dependencies and eliminating dead stores. The McNetKAT dialect of Probabilistic NetKAT can also be compiled to PRISM [27], though it also features a custom solver that outperforms PRISM on large networks.

Bayonet [6] is another recent language for analyzing stateful networks. It is more general than Netter, as even the ordering of network events is taken into ac-

count. Moreover, Bayonet programs can condition distribution parameters based on the observations they make, and then run Bayesian inference to determine those parameters. Unfortunately, the complexity of features modeled by the language poses great challenges for scalability. Bayonet requires users to bound the number of packets transmitted in the network and the number of network events that can occur. In the case studies analyzed by the authors, these numbers go to at most 20 packets and to a thousand network events (though in most case studies only a few tens of events are allowed). By contrast, by aggregating traffic at the level of flows, Netter analysis can scale to much larger time frames, since we can compute performance metrics for a network's long-term distribution. On the simple network of Section 4.1, we have seen that Netter can scale up to thousands of nodes, while prior work [6, 27] showed that Bayonet's analysis scales up to about 30 nodes (though, admittedly, on a somewhat different setup). On the other hand, Netter scalability degrades substantially when handling more complex networks, such as those of Section 4.3, which where limited to four servers. This issue seems challenging to address with Netter's current back-end: if each node in the network comprises 10 possible states, a modest size, the number of states of the system will be proportional to $10^{\#\text{nodes}}$ in the worst case.

6 Conclusion and Future Work

We have presented a framework for formally analyzing the probabilistic quantitative properties of networks. We showed the design and implementation of Netter, a language for verifying quantitative properties of stateful networks. Netter compiles its programs down to PRISM automata, and applies several optimizations to simplify their control flow, thus speeding up the analysis of the generated models. We evaluated Netter on a series of case studies. We observed that the tool scales up to sizable networks when reasoning about simple properties and routing schemes. We demonstrated how to use Netter to model more complex networks as well. Though the scalability of the analysis quickly degrades in these cases, we could still reason about important performance characteristics and use them to compare different routing strategies. In future work, we would like to address these scalability issues, potentially integrating symbolic analysis techniques that do not require explicitly enumerating the network state space.

Acknowledgments The authors would like to thank Steffen Smolka, Justin Hsu and Timon Gehr for useful discussions and clarifications. This work was partially funded by ONR award N000141812618, NSF award 1513961 and NSF award 1564009.

References

1. Abley, J., Lindqvist, K., Davies, E., Black, B., Gill, V.: IPv4 multihoming practices and limitations. RFC 4116, RFC Editor (July 2005), http://www.rfc-editor.org/rfc/rfc4116.txt

2. Anderson, C.J., Foster, N., Guha, A., Jeannin, J.B., Kozen, D., Schlesinger, C., Walker, D.: NetKAT: semantic foundations for networks. In: Jagannathan, S., Sewell, P. (eds.) The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014. pp. 113–126. ACM (2014), https://doi.org/10.1145/2535838.2535862
3. Braden, B., Zhang, L., Berson, S., Herzog, S., Jamin, S.: Resource reservation protocol (RSVP) – version 1 functional specification. RFC 2205, RFC Editor (September 1997), http://www.rfc-editor.org/rfc/rfc2205.txt
4. Fayaz, S.K., Yu, T., Tobioka, Y., Chaki, S., Sekar, V.: Buzz: Testing context-dependent policies in stateful networks. In: 13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16). pp. 275–289. USENIX Association, Santa Clara, CA (2016), https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/fayaz
5. Foster, N., Kozen, D., Mamouras, K., Reitblatt, M., Silva, A.: Probabilistic NetKAT. In: Thiemann, P. (ed.) Programming Languages and Systems - 25th European Symposium on Programming, ESOP 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings. Lecture Notes in Computer Science, vol. 9632, pp. 282–309. Springer (2016), https://doi.org/10.1007/978-3-662-49498-1_12
6. Gehr, T., Misailovic, S., Tsankov, P., Vanbever, L., Wiesmann, P., Vechev, M.T.: Bayonet: probabilistic inference for networks. In: Foster, J.S., Grossman, D. (eds.) Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018. pp. 586–602. ACM (2018), https://doi.org/10.1145/3192366.3192400
7. Henriksen, T.: VMCAI 2021 virtual machine (Sep 2020), https://doi.org/10.5281/zenodo.4017293
8. Hensel, C., Junges, S., Katoen, J.P., Quatmann, T., Volk, M.: The probabilistic model checker storm. CoRR **abs/2002.07080** (2020), https://arxiv.org/abs/2002.07080
9. Horn, A., Kheradmand, A., Prasad, M.: Delta-net: Real-time Network Verification Using Atoms. In: 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17). pp. 735–749. USENIX Association, Boston, MA (2017), https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/horn-alex
10. Hsu, J.: Probabilistic couplings for probabilistic reasoning. CoRR **abs/1710.09951** (2017), http://arxiv.org/abs/1710.09951
11. Jensen, J.S., Krogh, T.B., Madsen, J.S., Schmid, S., Srba, J., Thorgersen, M.T.: P-Rex: Fast verification of mpls networks with multiple link failures. In: Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies. CoNEXT '18 (2018)
12. Juniwal, G., Bjorner, N., Mahajan, R., Seshia, S.A., Varghese, G.: Quantitative network analysis. Tech. rep. (2016), http://cseweb.ucsd.edu/ varghese/qna.pdf
13. Kazemian, P., Chang, M., Zeng, H., Varghese, G., McKeown, N., Whyte, S.: Real Time Network Policy Checking Using Header Space Analysis. In: Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13). pp. 99–111. USENIX, Lombard, IL (2013), https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/kazemian

14. Kazemian, P., Varghese, G., McKeown, N.: Header space analysis: Static checking for networks. In: Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12). pp. 113–126. USENIX, San Jose, CA (2012), https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/kazemian
15. Khurshid, A., Zou, X., Zhou, W., Caesar, M., Godfrey, P.B.: VeriFlow: Verifying Network-Wide Invariants in Real Time. In: Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13). pp. 15–27. USENIX, Lombard, IL (2013), https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/khurshid
16. Kwiatkowska, M.Z., Norman, G., Parker, D.: PRISM 4.0: Verification of probabilistic real-time systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6806, pp. 585–591. Springer (2011), https://doi.org/10.1007/978-3-642-22110-1_47
17. Larsen, K.G., Schmid, S., Xue, B.: WNetKAT: A weighted sdn programming and verification language. In: 20th International Conference on Principles of Distributed Systems (OPODIS 2016) (2016)
18. Lee, S.B., Kang, M.S., Gligor, V.D.: CoDef: Collaborative defense against large-scale link-flooding attacks. In: Proceedings of the Ninth ACM Conference on Emerging Networking Experiments and Technologies. p. 417–428. CoNEXT '13, Association for Computing Machinery, New York, NY, USA (2013), https://doi.org/10.1145/2535372.2535398
19. Lopes, N.P., Bjorner, N., Godefroid, P., Jayaraman, K., Varghese, G.: Checking Beliefs in Dynamic Networks. In: 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15). pp. 499–512. USENIX Association, Oakland, CA (2015), https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/lopes
20. Mai, H., Khurshid, A., Agarwal, R., Caesar, M., Godfrey, P.B., King, S.T.: Debugging the Data Plane with Anteater. In: Proceedings of the ACM SIGCOMM 2011 Conference. pp. 290–301. SIGCOMM '11, ACM, New York, NY, USA (2011), http://doi.acm.org/10.1145/2018436.2018470
21. Mitzenmacher, M., Upfal, E.: Probability and computing : randomized algorithms and probabilistic analysis. Cambridge University Press, New York (2005)
22. Moggi, E.: Computational lambda-calculus and monads. In: Proceedings of the Fourth Annual Symposium on Logic in Computer Science (LICS '89), Pacific Grove, California, USA, June 5-8, 1989. pp. 14–23. IEEE Computer Society (1989), https://doi.org/10.1109/LICS.1989.39155
23. Panda, A., Lahav, O., Argyraki, K., Sagiv, M., Shenker, S.: Verifying reachability in networks with mutable datapaths. In: 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17). pp. 699–718. USENIX Association, Boston, MA (Mar 2017), https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/panda-mutable-datapaths
24. Pathak, A., Zhang, M., Hu, Y.C., Mahajan, R., Maltz, D.A.: Latency inflation with MPLS-based traffic engineering. In: Thiran, P., Willinger, W. (eds.) Proceedings of the 11th ACM SIGCOMM Internet Measurement Conference, IMC '11, Berlin, Germany, November 2-, 2011. pp. 463–472. ACM (2011), https://doi.org/10.1145/2068816.2068859

25. Smith, J.M., Schuchard, M.: Routing around congestion: Defeating DDoS attacks and adverse network conditions via reactive BGP routing. In: 2018 IEEE Symposium on Security and Privacy (SP). pp. 599–617. IEEE (2018)
26. Smolka, S., Kumar, P., Foster, N., Kozen, D., Silva, A.: Cantor meets Scott: Semantic foundations for probabilistic networks. In: Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages. p. 557–571. POPL 2017, Association for Computing Machinery, New York, NY, USA (2017), https://doi.org/10.1145/3009837.3009843
27. Smolka, S., Kumar, P., Kahn, D.M., Foster, N., Hsu, J., Kozen, D., Silva, A.: Scalable verification of probabilistic networks. In: McKinley, K.S., Fisher, K. (eds.) Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019. pp. 190–203. ACM (2019), https://doi.org/10.1145/3314221.3314639
28. Stoenescu, R., Popovici, M., Negreanu, L., Raiciu, C.: SymNet: Scalable symbolic execution for modern networks. In: Proceedings of the 2016 Conference on ACM SIGCOMM 2016 Conference. pp. 314–327. SIGCOMM '16, ACM, New York, NY, USA (2016), http://doi.acm.org/10.1145/2934872.2934881
29. Tarreau, W.: Test driving "power of two random choices" load balancing. https://www.haproxy.com/blog/power-of-two-load-balancing/ (Apr 2019)
30. Team, T.C.D.: The Coq Proof Assistant, version 8.12.0 (Jul 2020), https://doi.org/10.5281/zenodo.4021912
31. Tran, M., Kang, M.S., Hsiao, H.C., Chiang, W.H., Tung, S.P., Wang, Y.S.: On the feasibility of rerouting-based DDoS defenses. In: 2019 IEEE Symposium on Security and Privacy (SP). pp. 1169–1184. IEEE (2019)
32. Tschaen, B., Zhang, Y., Benson, T., Benerjee, S., Lee, J., Kang, J.M.: SFC-Checker: Checking the Correct Forwarding Behavior of Service Function Chaining. In: IEEE SDN-NFV Conference (2016)
33. Xie, G.G., Zhan, J., Maltz, D.A., Zhang, H., Greenberg, A., Hjalmtysson, G., Rexford, J.: On Static Reachability Analysis of IP Networks. In: INFOCOM 2005. 24th Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings IEEE. vol. 3, pp. 2170–2183. IEEE (2005)
34. Xue, L., Luo, X., Chan, E.W., Zhan, X.: Towards detecting target link flooding attack. In: 28th Large Installation System Administration Conference (LISA14). pp. 90–105 (2014)
35. Yang, H., Lam, S.S.: Real-Time Verification of Network Properties Using Atomic Predicates. IEEE/ACM Transactions on Networking **24**(2), 887–900 (4 2016). https://doi.org/10.1109/TNET.2015.2398197
36. Yuan, Y., Moon, S.J., Uppal, S., Jia, L., Sekar, V.: NetSMC: A custom symbolic model checker for stateful network verification. In: Bhagwan, R., Porter, G. (eds.) 17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020, Santa Clara, CA, USA, February 25-27, 2020. pp. 181–200. USENIX Association (2020), https://www.usenix.org/conference/nsdi20/presentation/yuan
37. Zeng, H., Kazemian, P., Varghese, G., McKeown, N.: Automatic test packet generation. IEEE/ACM Trans. Netw. **22**(2), 554–566 (4 2014), http://dx.doi.org/10.1109/TNET.2013.2253121
38. Zeng, H., Zhang, S., Ye, F., Jeyakumar, V., Ju, M., Liu, J., McKeown, N., Vahdat, A.: Libra: Divide and Conquer to Verify Forwarding Tables in Huge Networks. In: NSDI. vol. 14, pp. 87–99 (2014)

39. Zhang, H., Zhang, C., Azevedo de Amorim, A., Agarwal, Y., Fredrikson, M., Jia, L.: Netter: Probabilistic, stateful network models (Oct 2020), https://doi.org/10.5281/zenodo.4089060