ProbFlow: Using Probabilistic Programming in Anonymous Communication Networks

Hussein Darir
University of Illinois
Urbana-Champaign
hdarir2@illinois.edu

Geir Dullerud
University of Illinois
Urbana-Champaign
dullerud@illinois.edu

Nikita Borisov
University of Illinois
Urbana-Champaign
nikita@illinois.edu

***Abstract*—We present $ProbFlow$, a probabilistic programming approach for estimating relay capacities in the Tor network. We refine previously derived probabilistic model of the network to take into account more of the complexity of the real-world Tor network. We use this model to perform inference in a probabilistic programming language called *NumPyro* which allows us to overcome the analytical barrier present in purely analytical approach. We integrate the implementation of $ProbFlow$ to the current implementation of capacity estimation algorithms in the Tor network. We demonstrate the practical benefits of $ProbFlow$ by simulating it in flow-based Python simulator and packet-based Shadow simulations, the highest fidelity simulator available for the Tor network. In both simulators, $ProbFlow$ provides significantly more accurate estimates that results in improved user performance, with average download speeds increasing by 25% in the Shadow simulations.**

I. Introduction

Tor [9] is a system that helps protect online privacy and circumvent any censorship that may be present on the Internet. It has millions of daily active users [25]. Tor uses a network of volunteer *relays* to help encrypt users' traffic and obscure its source and destination. These relays have a wide range of capacities; hence, it's important to balance the load on them so that all users receive consistent service.

Although load balancing is a well-researched subject, it faces unique challenges because of Tor's security and privacy constraints. First, in order to protect their anonymity, Tor clients use source routing, which forgoes the use of a conventional feedback-based load-balancer and allows each client to choose which relays to use. Clients use a set of weights corresponding to the relays network capacity to stochastically choose relays. Second, by relying on relays to disclose their own capacities, as was the case with earlier versions of Tor, adversaries are given the opportunity to violate anonymity by inflating their nodes capacities in order to increase their traffic [4].

This was the main motivation behind the development of $TorFlow$ [20] and $sbws$ [18], two bandwidth scanning systems. Both systems use external probes to evaluate the performance of relays and use the bandwidth of those probes to adjust the value reported by the relay itself. However, the capacity estimates generated by both algorithms vary considerably over time and between different algorithm instances.

A new estimation mechanism, $MLEFlow$ [6], that performs maximum likelihood estimation using *multiple* measurements was recently proposed. This mechanism uses a simplified probabilistic model of the Tor network and finds the capacity value that maximizes the probability of observing the full history of measurements obtained for each relay.

While $MLEFlow$ was shown to achieve higher accuracy than the existing Tor capacity estimation, its probabilistic model simplifies away a number of important Tor features, and the computation of maximum likelihood required complex mathematical analysis. Our goal is to investigate whether an alternative method probabilistic estimation—*probabilistic programming*—can offer a superior approach for this context. Probabilistic programming allows the specification of probabilistic models using a programming language. It then performs *inference* to learn unknown parameters of the models based on a set of observations. In this paper, we first show how we can recast the $MLEFlow$ model as a probabilistic program, and then extend it to better capture the actual dynamics of Tor.

Expressing $MLEFlow$ flow as a probabilistic program is straightforward and requires only 10 lines of code. Moreover, our simulations show that the inference performed by probabilistic programming produces results that are at least as accurate as $MLEFlow$, demonstrating that the automated inference performed by probabilistic programming does not result in any accuracy tradeoffs. We next extend the $MLEFlow$ model, removing some of the simplifications that it introduced into the model of Tor. In particular, we model interaction between relays by modeling three-relay paths that share bandwidth, and we also model underloaded networks wherein clients do not fully utilize all of the relays' capacities. This model much better captures the dynamics of the actual Tor network, while using only 25 lines of code. We call the extended model $ProbFlow$.

We then perform extensive simulations comparing the existing Tor estimation algorithms—$TorFlow$ and $sbws$, $MLEFlow$ and $ProbFlow$ to show that the proposed algorithm results in significantly more accurate relay capacity estimates. We use a custom-built flow-based simulator, written in Python, to simulate the behavior of the Tor network under the different algorithms. We also simulate the different estimation mecha-

Network and Distributed System Security (NDSS) Symposium 2023
27 February - 3 March 2023, San Diego, CA, USA
ISBN 1-891562-83-5
https://dx.doi.org/10.14722/ndss.2023.24140
www.ndss-symposium.org

nisms in Shadow-2 [15], a high-fidelity simulation framework that runs the actual Tor C code and simulates network events at a packet level. The Shadow-2 simulation were performed in a cloud based parallel implementation of a scaled-down network. More details about the simulation baselines are presented in Section II-D.

Both types of simulations shows the increased accuracy of $ProbFlow$. In fact, the overall estimation error of $ProbFlow$ reaches below 10% for all classes of relays in the network, while the error is well above 50% for the other algorithms. This increased accuracy leads to better load balancing between users in the network. More than 80% of users witnessed improved download speeds when using the estimates of $ProbFlow$ compared to the currently deployed algorithms.

II. Background

In this section, we present the path allocation technique currently in use in the Tor network to select user *path* and demonstrate the importance of having accurate relay capacity estimates. We then explain how capacity estimation is done in Tor and the different proposed capacity estimation algorithms.

A. Path Allocation in Tor

Currently, the Tor network contains around 7000 *relays* [24] used for user traffic forwarding. In order to create a connection, a user chooses a *path* of three distinct relays to construct a circuit that forwards traffic in both directions. The entire path is only known by the user, while each relay knows only its predecessor and successor. Additionally, to obscure the correspondence between incoming and outgoing traffic, the traffic is encrypted / decrypted at each node.

Relays in Tor have heterogeneous capacities[1] that can differ by orders of magnitude (Figure 2). Relays are also divided into different classes with different capabilities: *exits* that can be used in the second or last position of a path, *guards* that can be used in the first or second position, *exit-guards* that can be used in any position, and *middles* that can be used in the second position [23]. We denote the corresponding classes of relays by e, g, d and m, respectively.

Each class of relays can be chosen to be in any of the three positions in a path according to a probability W: with a first subscript referring to the position in the user path and a second subscript referring to the class considered. As an example, the exits class e will have the triplet of probabilities (W_{ge}, W_{me}, W_{ee}) where W_{ge} refers to the probability that an exit flagged relay is used in the first position of the user path. The sum of the three probabilities of a given class will be equal to 1. Similarly, for class g we will have (W_{gg}, W_{mg}, W_{eg}); for class d, (W_{gd}, W_{md}, W_{ed}); and for class m, (W_{gm}, W_{mm}, W_{em}).

Since middle relays are selected in the second position only, then $W_{gm} = W_{em} = 0$ and $W_{mm} = 1$. On the other hand, exits relays can't be chosen in the first position, thus $W_{ge} = 0$, and guard relays can't be chosen in the last position i.e. $W_{eg} = 0$. The remaining seven probabilities $(W_{me}, W_{ee}, W_{gg}, W_{mg}, W_{gd}, W_{md}, W_{ed})$ are computed according to Section 3.8.3 of the Tor directory protocol [23] in order to balance bandwidth among classes. Because the exit relays are the most scarce relays in the Tor network and have the smallest total capacity of all the classes, the probabilities are computed as follows (Case 3, subcase b):

$$W_{ed} = W_{ee} = 1,$$
$$W_{md} = W_{gd} = W_{me} = 0,$$
$$W_{mg} = \frac{\sum_{k\in g} C[k] - \sum_{j\in m} C[j]}{2\sum_{k\in g} C[k]},$$
$$W_{gg} = 1 - W_{mg},$$

where we define $C[j]$ to be the estimated capacity of relay j. As can be noted, exits and exit-guards relays are only used for the last position of a path.

To create a path, the relays are sampled from the classes with a probability proportional to their estimated capacity. In the general case, the probability of choosing a relay $j \in N$ where N can be any of the 4 classes $\{e, g, d, m\}$ as the first node in a path is

$$w^g[j] = \frac{W_{gN} C[j]}{\sum_{k\in e} W_{ge} C[k] + \sum_{k\in g} W_{gg} C[k] + \sum_{k\in d} W_{gd} C[k] + \sum_{k\in m} W_{gm} C[k]}. \quad (1)$$

Notice that we only used the probabilities of each of the classes being in the first position of a path. Similarly for $w^m[j]$ and $w^e[j]$ in which we use the probability that a class is in the second and third position of a user's path respectively.

The intuition behind this path allocation approach is the fact that if the estimated capacities are equal to the true capacities, that we call $C^*[j]$, the expected number of paths using each relay will be proportional to its bandwidth. In this paper, we use $X[j]$ to denote the number of paths using relay j.

B. Security Considerations

Since capacity estimation is used as input for Tor's path selection process, it is important to consider its security and privacy properties. If the path chosen by the user consists of relays controlled by an adversary, then the attacker will have full knowledge of the path and render Tor's anonymity protection completely ineffective. In fact, using timing analysis, it is enough to watch the incoming and outgoing traffic of the first and last Tor relays [30] to link a user with a destination. Therefore, if the capacity estimate gives a higher weight to the adversary's selected relays, this increase the likelihood of anonymity being compromised.

A relay itself has the most visibility into its own network capacity, but a compromised relay can simply lie and claim to have more capacity than it actually does without deploying high-bandwidth relays [4]. Another way to determine the network's capacity is to use probes. This is the approach we take in this paper, but it does not preclude the possibility of an attack. A relay may recognize probe circuits and treat them preferentially which will result in the overestimation of its estimate [26]. Alternatively, an attacker can predict when

[1]By "capacity" we refer to the smaller of upload and download bandwidth limit on the relay. In some cases, other bottlenecks may exist on the path between two relays but a per-node bandwidth limit is a common and useful model of network capacity constraints.

a particular relay will be probed and perform a denial of service on the relay or the prober to artificially reduce that estimate [16]. The later attack can be addressed by using randomization, while the former is harder to address but we are optimistic that censorship circumvention research that aims to prevents identification of undesirable types of traffic can find a solution to this problem.

C. Relays Capacities Estimation in Tor

The currently deployed capacity estimation algorithm in the Tor network is called $TorFlow$. It has been in use since 2012. $TorFlow$ uses two types of measurements: the *self-reported bandwidth* and the *measured bandwidth*. To determine the self-reported bandwidth, each relay computes the maximum sustained download and upload bandwidth over a 5-second period over the last 5 days. This value is then reported to Tor *directory authorities* and we will use $b_t[j]$ to refer to the self-reported bandwidth of relay j reported at time t.

The self-reported bandwidth is then adjusted based on the results of the measured bandwidth that we denote $m_t[j]$ for a relay j at time t. The directory authorities create a probe through each relay and download a file measuring the realized bandwidth of the probe. The adjustment is computed by first calculating the average measured bandwidth across all relays, μ_t, and then multiplying the self-reported bandwidth by the ratio of the measured bandwidth and the average: $C^A_{t+1}[j] = b_t[j]m_t[j]/\mu_t$. The authorities compute the estimates of all the relays in the network and distribute the information to the clients in a *consensus* document, published every hour.

However, this version of capacity estimation has a number of disadvantages [6]. An under-loaded relay will underestimate its self-reported bandwidth, which leads to a small estimated capacity, which in turn leads to low-load, and hence, low self-reported bandwidth. This problem leads to the well-documented ramp-up period of new relays [7]. Another problem is the use of self-reported bandwidth which creates the opportunity for a low-resource attack on the Tor network [4]. In particular, a relay can publish a high self-reported bandwidth for itself, which is likely to result in high consensus weight [17] and hence will cause more clients to choose the relay and create more chances for it to break users anonymity.

Another version of $TorFlow$ used the previous estimated capacity instead of the self-reported bandwidth: $C^{TF}_{t+1}[j] = C^{TF}_t[j]m_t[j]/\mu_t[C]$. We denote this version as $TorFlow$-P. The intuition behind this algorithm is similar to the original $TorFlow$: if the current estimated capacity of a relay is too high it will have a below average performance. This is depicted by its measured bandwidth being less than the average across all relays, and thus its estimated capacity will be adjusted down. Tor switched away from this approach because, when deployed, the feedback mechanism allowed the weights to significantly deviate from network capacities.

Another currently deployed capacity estimation algorithm is called $sbws$. It is the result of recent effort to upgrade and re-emgineer $TorFlow$. As reported in [6], $sbws$ uses the minimum of the self-reported bandwidth and the previous estimated capacity: $C^S_{t+1}[j] = \min(C^S_t[j], b_t[j])m_t[j]/\mu_t$. While this version is still susceptible to under-weighting relays with low self-reported bandwidth, it is more resilient to high self-reported bandwidth.

As of this writing, several Tor bandwidth authorities use $TorFlow$ and $sbws$ to compute the capacity estimates of relays in the Tor network. The consensus bandwidth published will be the median of the estimates of all the authorities.

A new estimation algorithm was developed in $MLEFlow$ [6] that takes into account the whole history of measured bandwidth instead of the last value. $MLEFlow$ showed that $TorFlow$-P is actually equivalent to a maximum likelihood estimation of the capacity of a relay using the last value of the measured bandwidth of the relay. $MLEFlow$ maximized the probability of observing the full history of measured bandwidth given the weights that were used when generating those measurements using a simplified model of the Tor network. We will go into more details of the model used in Section III-A. The paper [6] also proposed a closed form approximation to compute the capacity estimate of a relay j:

$$C^{MF}_{t+1}[j] \approx \exp\left(\frac{\sum_{i=0}^{t} \frac{1}{m_i[j]} \log(m_i[j]\lambda_s w_i[j])}{\sum_{i=0}^{t} \frac{1}{m_i[j]}}\right).$$

While $MLEFlow$ showed promise in estimating capacity with higher accuracy than the currently deployed algorithms, its simplified model of the Tor network works best for exit-flagged relays and had relatively higher errors for other classes of relays. Another shortcoming of the model is that it failed to depict other bandwidth constraints that can be present in the network.

D. Simulation Baselines

The Shadow simulation framework [13] is a state-of-the-art discrete event simulator that is commonly used to study the Tor network. Shadow runs the actual implementation of Tor, which simulates a number of relays running on a single host, communicating over a custom-built simulated network. Shadow has been used to analyze various properties of Tor, as well as potential improvements.

The new version of Shadow [15] is able to run Python which is used to implement $TorFlow$ and $sbws$. Since the $sbws$ implementation [22] supports a configurable weight computation algorithm, termed *scaling* in its implementation; we added $MLEFlow$ and $ProbFlow$ as new scaling algorithms. To better model $TorFlow$ and $sbws$, we have used an idealized self-reported bandwidth that is equal to the actual (ground truth) capacity of the relay. In the real-world, the value of the self-reported bandwidth takes several days to stabilize [7] which is impractical to simulate in Shadow. In fact, the self-reported bandwidth is intended to capture the true capacity of a relay but at times fall short due to periods of low load. Thus taking its value to be the actual capacity of the relay captures a "best-case" scenario for $TorFlow$ and $sbws$. We will denote algorithms using this value by adding a * to the algorithm name.

Shadow provides highly realistic simulation as has been validated in previous studies [12]. However, this technique is resource-intensive and can only simulate a fraction of the Tor network. As an example, the simulations of Section VI are

scaled to 3% of the actual number of relays in the Tor network and run approximately 24 times slower than real-time, despite using a 48-core Azure virtual machine with 384 GiB of RAM.

To simulate a 100% network, we implement a simpler version of the Tor network in which each client stream is modeled as a flow and we use max-min fairness to divide bandwidth between the flows (described in more details in Section IV). Since the flow-based simulator does not capture the complexity of a real-world network, it can simulate an entire Tor network with millions of clients in Python on a desktop computer. However, this simulator still captures essential behavior of Tor.

The changes in load and ramp-up effects that cause the self-reported bandwidth to change over time are not captured in the flow-based model and hence we use the idealized self-reported bandwidth to simulate $TorFlow$ and $sbws$.

III. *MLEFlow* as a Probabilistic Program

In this section, we introduce the notion of probabilistic programming by presenting the implementation of the probabilistic model of $MLEFlow$ [6] as a probabilistic program.

A. $MLEFlow$ Probabilistic Model for Capacity Estimation

Many modern applications (e.g., in machine learning, robotics, autonomous driving, medical diagnostics, and financial forecasting) need to make decisions under uncertainty. Probabilistic programming is a software-driven method for creating probabilistic models and then using them to make inferences.

In order to demonstrate the application of probabilistic programming to the problem of relay capacity estimation, we will explain the implementation of a probabilistic program for the simplified model derived in $MLEFlow$ [6]. We will later refine the model to add more of the complexity encountered in the actual Tor network.

The $MLEFlow$ model related the unknown capacity of a relay j, which we will denote as $C^*[j]$, to the observed bandwidth of a measurement probe through the relay j at time t, $m_t[j]$. The key assumption behind $MLEFlow$ is that the capacity of the relay is evenly split among all the flows that are using the relay j at time t. This includes the one measurement flow and a number of user flows, which we will denote by the random variable $X_t[j]$. $MLEFlow$ models user flows as arriving with a Poisson process with rate λ_s. Since Tor clients choose relays using weights published by the directory authorities, we can say that $X_t \sim \text{Pois}(\lambda_s w_t[j])$ where $w_t[j]$ is the weight assigned to relay j at time t. We can therefore express the relationship between the unknown capacity and the observations:

Definition 1 ($MLEFlow$ model): The measurement of any relay $j \in [n]$ at any time $t \in \mathbb{N}$, can be written as:

$$M_t^{MF}[j] = \frac{C^*[j]}{X_t[j]+1}, \tag{2}$$

where we use the superscript MF to identify $MLEFlow$.

In $MLEFlow$ [6], the actual capacity $C^*[j]$ was considered an unknown scalar that needed to be estimated. The estimation was done manually by maximizing the log likelihood of the probability of observing a certain measurement over a defined bounded capacity set $\kappa \subset \mathbb{R}^n_{\geq 0}$. We next show how to estimate $C^*[j]$ using probabilistic programming.

To do this, we consider $C^*[j]$ as a *latent variable* that follows some unknown parametric distribution. We then use a probabilistic programming language (PPL) to specify the model, relating the latent variable to observed samples. Then the probabilistic programming framework uses Stochastic Variational Inference (SVI) to estimate the distribution parameters.

In our example, we model $C^*[j]$ as a Weibull distribution. Weibull has the requisite property that it has support among non-negative real numbers. Additionally, as the shape parameter in the Weibull distribution goes to infinity, the distribution converges to the Dirac delta function centered at the scale parameter. We thus expect that as the model makes use of more measurements, the shape parameter will increase producing a more confident estimate of the capacity. This is captured in a *guide function* of the probabilistic programming language, as shown in Algorithm 1.

Algorithm 1 Guide

1: **input:** $\lambda_s, w_{[t]}[j], m_{[t]}[j], \gamma_0$.
2: **parameter 1:** scale
3: **parameter 2:** shape
4: $C \xleftarrow{\text{sampled}}$ Weibull(scale, shape)

We also must define a prior distribution for the random variable $C^*[j]$. We choose an exponential distribution, as it matches the empirical data gathered about Tor relays by past measurements. Then we specify the *model* that relates the observed measurements to the latent variable. Our model is simply programmatic expression of (2), as shown in Algorithm 2. In general, the model can be any Python program that relates the sample of the latent variable from the prior distribution to the samples corresponding to the observations. Note, particularly, that our model in 2 uses a for loop to be able to capture multiple observations from multiple time points, which was a key innovation of $MLEFlow$. Each measurement is treated as a sample from a distribution that is conditioned on the observed value C.

Algorithm 2 $MLEFlow$ model

1: **input:** $\lambda_s, w_{[t]}[j], m_{[t]}[j], \gamma_0$.
2: $C \xleftarrow{\text{sampled}} exp(\gamma_0)$
3: **for** $i \in [0, .., t]$ **do**
4: $\quad o \xleftarrow{\text{sampled}} \frac{C}{Pois(\lambda_s w_i[j])+1}$ given the observed value $m_i[j]$.

The probabilistic programming language that we are going to use in this paper is NumPyro. NumPyro is a probabilistic programming language built on Python, PyTorch and Jax. NumPyro programs are just Python programs that support a number of inference algorithms, such as SVI. The Python code for our models is shown in Appendix A. The output of the SVI applied on the model will be estimates of the shape and scale parameters of the Weibull distribution. We then use those

estimates to sample the estimated capacity of relay j from the Weibull distribution.

What makes the model simple—only 10 lines of Python for both the model and the guide—is that we only specify the "forward" direction: how the measurement samples are distributed given the latent variable values. SVI then performs the "backward" direction of inferring the latent variable given the measurements. This replaces the complex mathematical derivation of maximum likelihood estimator for a transformed Poisson distribution that was used by Darir et al. in MLEFlow. And, as we will show in Section V, we are able to obtain estimates that are no less accurate than using the derivations by Darir et al.

Moreover, the use of probabilistic programming allows us to add more complexity to the model to better capture the behavior of Tor. To make their analysis tractable, Darir et al. used a greatly simplified Tor model, in particular assuming that bandwidth is divided equally among all flows on a relay. We next show how to extend our probabilistic programming model for MLEFlow to model interactions across relays, which are a key component of the Tor network.

IV. *ProbFlow* Refined Probabilistic Model

In this section, we will work on deriving a new probabilistic model relating the measurement at each epoch to the actual capacity of a relay. We will do that by relaxing and refining some of the assumptions considered in [6]. We will then discuss the implementation of the new probabilistic model that we derived in NumPyro.

A. Full three relays probabilistic model: ProbFlow

The new probabilistic model derived in this paper will keep assumptions 2 and 3 of $MLEFlow$ presented in Section III-A. Notably, we still consider a synchronous model where user arrival follows a Poisson distribution with rate λ_s. However assumption 1 will be relaxed. We consider a model where relays fall into 4 classes, (e, g, d, m), and each user path will go through three relays.

As discussed in Section II-A, we consider the case where the relays of a given class can be selected to be in any position in a path according to a probability. Class e will have the triplet of probabilities (W_{ge}, W_{me}, W_{ee}), class g will have (W_{gg}, W_{mg}, W_{eg}), class d will have (W_{gd}, W_{md}, W_{ed}), and class m will have (W_{gm}, W_{mm}, W_{em}).

The total number of paths passing through the j^{th} relay at the t^{th} epoch, is still a random variable, $X_t[j]$, with distribution $\text{Pois}(\lambda_s w_t[j])$ where $w_t[j] = w_t^g[j] + w_t^m[j] + w_t^e[j]$ with

$$w_t^g[j] = \frac{W_{gN} C_t[j]}{\sum_{k\in e} W_{ge} C_t[k] + \sum_{k\in g} W_{gg} C_t[k] + \sum_{k\in d} W_{gd} C_t[k] + \sum_{k\in m} W_{gm} C_t[k]}, \quad (3)$$

representing the probability that j is chosen in the first position of a user's path where $N \in e, g, d, m$ is the class of relay j. Similarly $w_t^m[j]$ and $w_t^e[j]$ are computed using the probabilities of each class being in the second and third position, respectively.

In this section, in order to derive the model we consider a target relay, r_1, that belongs to class $N_1 \in \{e, g, d, m\}$.

We first examine the case where r_1 is selected in the first position of a user's path. For all relays in $[n] \setminus \{r_1\}$, the weight that a relay r_2 is chosen in the second position is equal to the product of the probability that the relay's class, $N_2 \in \{e, g, d, m\}$, is used in the second position, i.e W_{mN_2}; and the estimated capacity of relay r_2 at epoch t, $C_t[r_2]$. Similarly, the weight that a relay $r_3 \in [n] \setminus \{r_1, r_2\}$ is chosen in the third position is $W_{eN_3} C_t[r_3]$, where N_3 is the class of relay r_3.

Using the same logic, if the target relay r_1 is selected in the second position of a user's path, the weight that a relay $r_2 \in [n] \setminus \{r_1\}$ is chosen in the first position of the path is $W_{gN_2} C_t[r_2]$ and r_3 in the third position of the path will be $W_{eN_3} C_t[r_3]$. Finally, if r_1 is selected in the third position, the weight of r_2 in the first position will be $W_{gN_2} C_t[r_2]$ while r_3 in the second position will have a weight of $W_{mN_3} C_t[r_3]$.

Thus, given the total number of paths using the target relay r_1, we can find the conditional probability that a pair (r_2, r_3) is chosen as the two other relays in the paths, $Pr((r_2, r_3)|X_t[r_1])$, by multiplying the weights of both relays and normalizing it by the sum of weights of all possible combinations of pairs for all possible positions of r_1.

Now that we have a probability distribution over the pair of relays that can be chosen alongside r_1 in a user's path, we can express the measurement of r_1 in terms of those probabilities by adding a further assumption to the model.

B. Integrating Max-Min Fairness Bandwidth Allocation in ProbFlow Model

We assume that the bottleneck capacity of a relay j at a given epoch t is equal to the measurement of the relay during this epoch, $m_t[j]$. Hence, the bottleneck of a pair (r_2, r_3) will be the minimum of the observations of the two relays.

We use the max-min fairness bandwidth allocation algorithm presented in [5]. Notably, the observation of each relay will be equal to the ratio of its *residual capacity*, *Cres*, over the norm of its *residual paths*, *Rres*. Where *residual capacity* is equal to the actual capacity of the relay from which we subtract the bandwidths of all paths passing through it that are already bottlenecked at other relays. And *residual paths* is the set formed by the observation probe and the paths passing through the relay after removing those paths whose bandwidths are already allocated. Thus for the target relay r_1,

$$M_t^{PP}[r_1] = \frac{Cres[r_1]}{|Rres[r_1]|}, \quad (4)$$

where we use the supersctipt PP to identify *Probabilistic programming*.

A path going through r_1 and the pair (r_2, r_3) will be bottlenecked at r_2 or r_3 if the bottleneck of the pair is less than the measurement of r_1. Let B_{r_1} be the set of pairs (r_2, r_3) such that $m_t[r_1] > \min(m_t[r_2], m_t[r_3])$.

For each pair in B_{r_1}, we know the probability of the pair given the total number of paths using the target relay r_1. Thus for a pair (r_2, r_3), the expected number of paths going through the pair and r_1 will be equal to $X_t[r_1] Pr((r_2, r_3)|X_t[r_1])$. Thus, the total number of paths using r_1 but are bottlenecked at other relays is equal to

$\sum_{(r_2,r_3)\in B_{r_1}} X_t[r_1]Pr((r_2,r_3)|X_t[r_1])$. Moreover, the number of paths passing through r_1 after removing the paths bottlenecked at other relays is

$$|Rres[r_1]| = X_t[r_1]H_t^1[r_1] + 1 \tag{5}$$

where $H_t^1[r_1] = \left(1 - \sum_{(r_2,r_3)\in B_{r_1}} Pr\big((r_2,r_3)|X_t[r_1]\big)\right)$ and the additional one represents the observation probe.

We also know the bottleneck of each of the pairs in B_{r_1}, and we can find the total bandwidth of paths using r_1 but that are bottlenecked at the other relays to be $\sum_{(r_2,r_3)\in B_{r_1}} X_t[r_1]Pr((r_2,r_3)|x_t[r_1])\min(m_t[r_2], m_t[r_3])$. Thus,

$$|Cres[r_1]| = C^*[r_1] - X_t[r_1]H_t^2[r_1] \tag{6}$$

where $H_t^2[r_1] = \sum_{(r_2,r_3)\in B_{r_1}} Pr\big((r_2,r_3)|X_t[r_1]\big)\min(m_t[r_2], m_t[r_3])$.

$H_t^1[r_1]$ and $H_t^2[r_1]$ are two parameters that can be computed using the measurements of the relays in the network, the estimated capacities of the relays at each epoch and the triplet of probabilities of each class of relays.

Accordingly, we can write the measurement of a relay $j \in [n]$ at epoch t, as a function of the two parameters aforementioned, the actual capacity of the relay and the random variable $X_t[j]$.

Theorem 1 (ProbFlow Model): For any $j \in [n]$ and $t \in \mathbb{N}$, the measurement is

$$M_t^{PP}[j] = \frac{C^*[j] - X_t[j]H_t^2[j]}{X_t[j]H_t^1[j] + 1}, \tag{7}$$

where we use the supersctipt PP to identify *Probabilistic programming*.

Now that we derived the new probabilistic model we are going to use, we will present the implementation of *ProbFlow* in *NumPyro*.

C. ProbFlow Implementation Using NumPyro

Since we will still be using a Weibull distribution as an approximation of the posterior distribution, the guide part of our implementation will remain unchanged.

The model will take $H_{[t]}^1[j]$ and $H_{[t]}^2[j]$ as additional inputs and will implement Equation 7 for the observed random variable. Algorithm 3 presents the model implementation. The higher level implementation is shown in Figure 1.

Algorithm 3 *ProbFlow* model

1: **input:** $\lambda_s, w_{[t]}[j], m_{[t]}[j], \gamma_0, H_{[t]}^1[j], H_{[t]}^2[j]$.
2: $C \xleftarrow{\text{sampled}} exp(\gamma_0)$
3: **for** $i \in [0, .., t]$ **do**
4: $\quad o \xleftarrow{\text{sampled}} \frac{C - Pois(\lambda_s w_i[j])H_t^2[j]}{Pois(\lambda_s w_i[j])H_t^1[j]+1}$ given the observed value $m_i[j]$.

The Python code for this version of the model is presented in Appendix A. Note that this model is only slightly more complex than the MLEFlow model, with 25 lines of Python code. At the same time, direct mathematical analysis of this model would be highly complicated by the fact that the observations are based on a ratio involving two different Poisson distributions.

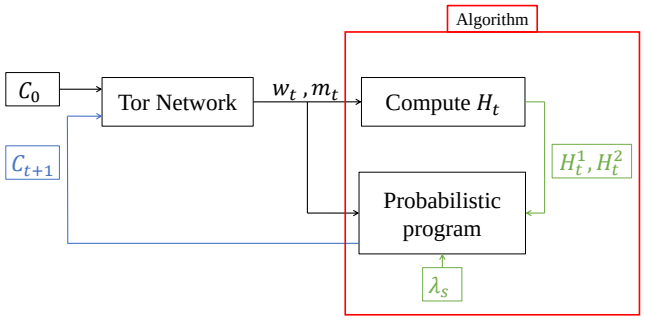


Fig. 1: Algorithm implementation.

D. Taking Underloaded Networks Into Consideration in ProbFlow Model

An additional layer of complexity that we can add to our model is client side bandwidth constraints. We assumed in our previous derivation that clients can utilize arbitrary amounts of bandwidth and are only bottlenecked by Tor relays. In general, client demand on Tor is lower than the overall available bandwidth. *MLEFlow* algorithm did not model these types of constraints in its model which affected the accuracy of its estimates in underloaded networks.

We assume that the average client demand, $Client_{avg}$ in KB/s is known. The only change to the model derivation of section IV-B is the fact that the bottleneck of a pair (r_2, r_3) will now be the minimum of the observations of the two relays and $Client_{avg}$. This change will only affect the computation of $H_t^2[r_1]$ which will now be

$$H_t^2[r_1] = \sum_{(r_2,r_3)\in B_{r_1}} Pr\big((r_2,r_3)|X_t[r_1]\big) \min(m_t[r_2], m_t[r_3], Client_{avg}). \tag{8}$$

The rest of the implementation will remain unchanged. We compare the performance of the different algorithms for the case of underloaded network in Section V. To implement user side constraints in flow-based simulations, we model the constraint as a fourth relay added to each user's path. The capacity of the fourth relay will be picked uniformly from an interval $[Client_{\text{min}}, Client_{\text{max}}]$ with mean equal to $Client_{avg}$.

V. Flow-Based Simulation

To compare the performance of the different Tor capacity estimation algorithms and the proposed method, we evaluate them using flow-based simulation of the Tor network. Those simulations leave out features like circuit construction, flow control and congestion and are implemented in Python. We evaluate the new *ProbFlow* algorithm as well as *MLEFlow*, *TorFlow-P* and *sbws*. For the purpose of this paper, we

use an idealized version of the self-reported bandwidth when simulating $sbws$, and we call this version $sbws^*$.

We use two metrics to evaluate the performance of the different algorithms:

1) Accuracy of the relay capacity estimates,
2) Bandwidth allocation distribution over users paths when using the capacity estimates generated.

The simulation algorithm used is similar to [6]. The inputs of the algorithm will be: the Poisson arrival rate λ_s, the total number of epochs T to be simulated, the *method* used for capacities estimation from *TorFlow-P, sbws*, MLEFlow* and *ProbFlow*. We also add an indicator *underloaded* $\in \{0, 1\}$ for underloaded network, and lower and upper bounds on clients side constraints as described in section IV-D. The algorithm will then output the bandwidth allocated to each user path as well as the estimated capacities and the weight vectors published over all the epochs simulated. The algorithm is presented in Algorithm 4.

At each epoch, $i \in [0, ..., T]$, the simulation algorithm samples the total number of users paths N_i joining the network from a Poisson distribution with rate λ_s (line 3). In line 4, N_i three relays paths are constructed using the weight vector w_i computed using the previous epoch estimated capacities. If *underloaded* is true (i.e. 1), a fourth relay is added to each user path with a capacity sampled uniformly from the interval $[Client_{\text{min}}, Client_{\text{max}}]$ in line 5. To generate the observation vector m_i, the algorithm uses max-min fairness bandwidth allocation algorithm [5] in line 6. After computing the estimated capacities using *method*, it deletes all the paths in the network and starts a new epoch.

Algorithm 4 Low fidelity simulation

1: **input:** λ_s, T, $method \in \{TorFlow\text{-}P, MLEFlow, sbws^*, ProbFlow\}$, $underloaded \in \{0, 1\}$, $Client_{\text{min}}, Client_{\text{max}}, w_0$.
2: **for** $i \in [0, ..., T]$ **do**
3: Pick the number of users $N_i \sim Poi(\lambda_s)$.
4: Construct users paths of three relays using w_i.
5: If $underloaded$, add a 4^{th} relay to each path with a capacity uniformly picked from $[Client_{\text{min}}, Client_{\text{max}}]$.
6: Compute m_i using max-min bandwidth alloc.
7: Compute w_{i+1} based on m_i and w_i using $method$.
8: Delete all paths in the network.
9: **return:** $m_{0:T}, w_{0:T}$

We consider a network similar to the current Tor network as of May 2022. The network has 7054 relays in total, with 360 exit flagged relays, 2910 guard flagged relays, 1190 exit and guard flagged relays and 2594 middle relays. As done by Traudt at al. [27], we use the highest reported observed bandwidth over a period of a month to be the ground truth capacity of the relays in our simulations. The distributions of the relays' capacities are shown in Figure 2.

We simulate each of the algorithms for $T = 10$ and $\lambda_s = 10^6$. For the case of underloaded network, we consider the clients interval $[Client_{\text{min}}, Client_{\text{max}}] = [50, 80]$ KB/s. Since the average clients' bandwidth for the full utilization case was

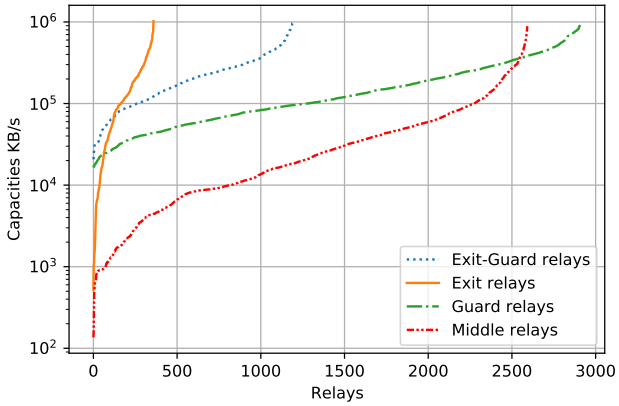


Fig. 2: Relays capacity distribution.

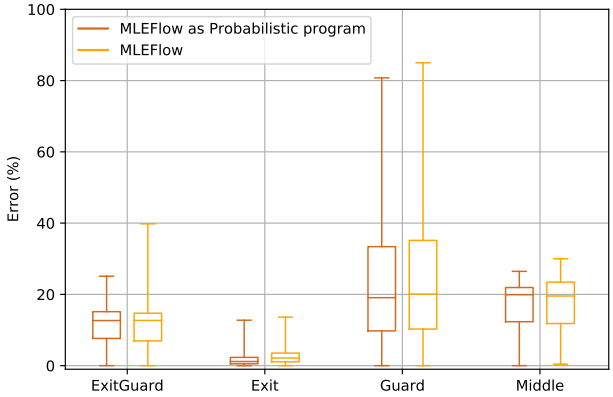


Fig. 3: Relays capacity estimation using the *MLEFlow* and the probabilistic program of *MLEFlow* in a fully utilized network scenario after 20 epochs of a 100% network.

around 100 KB/s, the cap means that a flow can utilize at most about 65% of the network capacity.

The probabilistic program version of *MLEFlow* results in identical error distribution as the original *MLEFlow* algorithm. We tested *MLEFlow* and its probabilistic programming implementation on the 100% network of Figure 2 in a fully utilized scenario (i.e. *underloaded* $= 0$). We plot the error distribution between the estimated capacities of the two algorithms and the actual capacities in Figure 3. Both algorithms resulted in approximately identical error distributions where the average error for exit relays stayed below 5% while the average error for the guard and middle classes of relays was around 20%. As discussed in [6], exit relays are expected to be the bottlenecks of the paths since they have the smallest total capacity of all the classes and that is why the model derived in *MLEFlow* works better for exit relays.

ProbFlow **results in more accurate relays' capacity estimates for all classes of relays.** Figure 4 as well as Table I show the error distribution in the same 100% network for a full utilization scenario (*underloaded* $= 0$). *ProbFlow* preserved an average error of less than 3% for all classes of relays. As

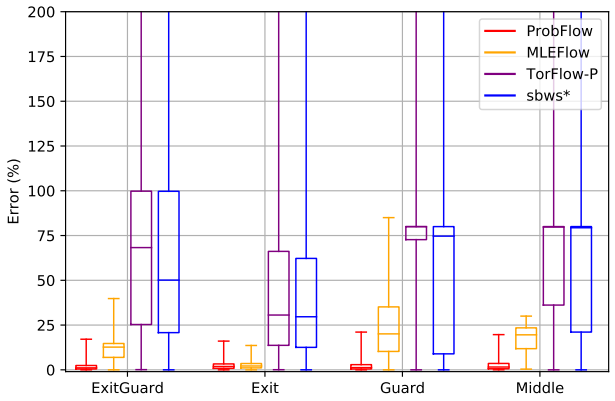


Fig. 4: Relays capacity estimation error distribution using the 4 different algorithms in fully utilized network scenario after 20 epochs of a 100% network.

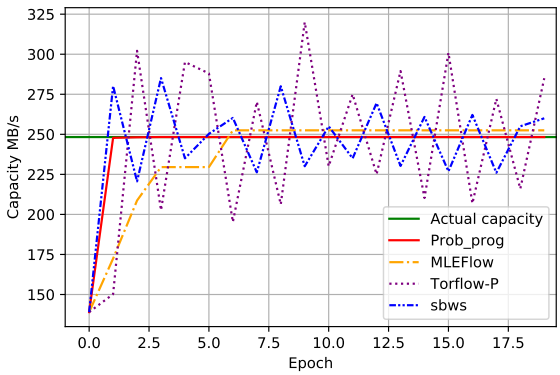


Fig. 5: The estimated capacity of an exit relay joining the network after 10 measurement epochs and staying for 20 epochs in MB/s.

discussed before, *MLEFlow* had an average error of less than 5% for exit relays while the average error for the guard and middle classes was around 20%. *TorFlow-P* estimates had an average error of 53% for exit relays, 71% for guard relays and 60% for middle relays. For *sbws**, the average error was around 50% for exit and guard relays and 60% for middle relays. It is clear that *ProbFlow* outperforms all the other algorithms in estimating the capacity of all relays' classes.

Using *ProbFlow* estimates results in higher and fairer bandwidth allocation between users. The distribution of the bandwidth allocated to 1 million users using the estimates of each algorithm is presented in Table I. The means of the bandwidths allocated for paths using the actual capacities and *ProbFlow* estimates are around 100 KB/s. The bandwidth allocated to paths when using *MLEFlow* estimates is lower at 81 KB/s. While for *sbws**, the mean is much lower at 66 KB/s and for *TorFlow-P* is significantly lower at 47 KB/s. Another advantage to the method developed is the fact that the bandwidth allocated are more fairly distributed around the mean as can be depicted by the smaller standard deviation of the bandwidth in Table I.

ProbFlow **handles joining relays better than the other algorithms.** We present the results of estimating the capacity of a relay that joins the network after 10 epochs of capacities estimation using the different algorithms in Figure 5. *ProbFlow* estimate converged to the actual capacity in one epoch, while *MLEFlow* estimate took around 5 epochs to converge. *TorFlow-P* and *sbws** estimates oscillated around the true capacity with large error even after 20 epochs.

ProbFlow **performs better than other algorithms in the underloaded network case.** We also simulated the 100% network of Figure 2 for the case of underloaded network (i.e. $underloaded = 1$) with a client constraint interval $[50, 80]$ KB/s. The results are shown in Figure 6. The average error of *MLEFlow* estimates for all classes of relays increases significantly: it increased to 10% for exit relays and jumped to above 40% for the other three classes. The estimates error also increased significantly for *TorFlow-P* and *sbws**. However the average error of *ProbFlow* estimates remained below 5% for all classes of relays.

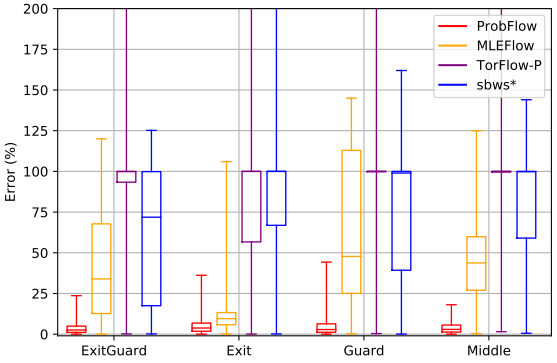


Fig. 6: Relays capacity estimation using the 4 different algorithms in an underloaded network scenario after 20 epochs of a 100% network.

VI. High-Fidelity Packet-Based Simulations

We use Shadow-2 [15], the most popular and validated platform for Tor experimentation, to compare the performance of the different capacity estimation algorithms. Shadow is a conservative-time discrete-event network simulator: it simulates hosts, processes, threads, TCP, and other kernel operations. Particularly, it runs the actual C implementation of Tor relays in a simulated network.

We configured 5 different 3% networks using the TorNetTools [15] and the Tor network data files (i.e., hourly network consensus and daily relay server descriptor files) of the month of May 2022. Each network contained around 195 Tor relays: 12 exit flagged, 61 guard flagged, 14 exit-guard flagged and 108 middle relays. On average, the total bandwidth of the relays is 1.4 GB/s, split into 79 MB/s for the exits, 730 MB/s for the guards, 288 MB/s for exits-guards and 297 MB/s for middle relays. We simulate 2000 clients, 3 directory authorities and a bandwidth authority.

Each of the simulated clients maintains a single file download for the duration of the epoch. At the end of the epoch, all streams are dropped and then restarted in order for the

TABLE I: Low-fidelity simulation results done in Python3. The results presented are for fully-utilized networks. *Est. method* is the method we use to update the weight vector in each epoch. *stats* are the statistics that we report for each simulation run. *relays cap. est. error* are the estimation errors for each class of relays. It is computed as : $error = \frac{|w_t - w^*|}{w^*} \times 100$. *paths bw* are the bandwidths allocated for users when the weight vectors are updated using different methods. The reported results are for the 20^{th} epoch.

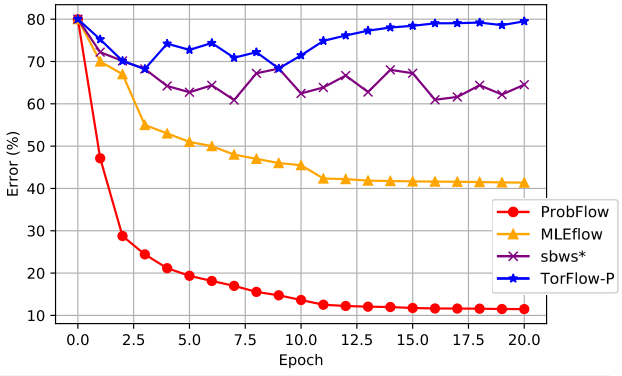
Est. method	stats	relays cap. est. err. (%) exitguard	exit	guard	middle	paths bw (KB/s)
Actual	mean	-	-	-	-	99.72
	std.	-	-	-	-	607.79
	max	-	-	-	-	291610
	min	-	-	-	-	0.06
TorFlow-P	mean	62.09	53.39	71.26	60.62	46.82
	std.	38.25	87.69	27.06	34.60	1526.3
	max	234.91	273.75	257.44	254.62	201840
	min	0	0	0	0	0.059
sbws*	mean	58.61	50.80	50.95	57.90	66.24
	std.	42.13	77.85	36.20	38.58	1286.18
	max	245.44	882.40	639.23	243.35	201840
	min	0	0	0	0	0.055
MLEFlow	mean	12.36	2.52	24.28	21.84	80.59
	std.	7.96	1.85	19.03	12.14	804.16
	max	39..80	13.60	85.21	64.77	256583
	min	0	0	0	0	0.054
ProbFlow	mean	1.80	2.29	2.15	2.44	100.54
	std.	2.04	1.90	2.55	2.67	652.01
	max	17.09	16.04	20.08	19.66	276534
	min	0	0	0	0	0.059

circuits generated to use the latest consensus file. We also set the duration of each epoch to be 10 minutes.

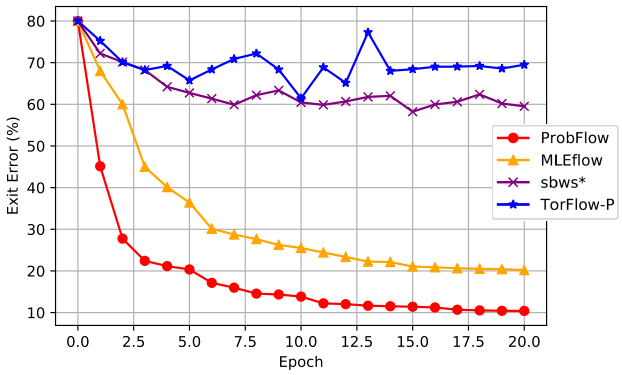
To run *sbws** and *TorFlow-P*, we use the Python code provided in [22]. As discussed in Section II-D, we have adapted the Python implementation of those algorithms [22] to simulate *MLEFlow* and *ProbFlow*. For *ProbFlow*, at each epoch, the authority reads the measurement of each relay; computes the two parameters, H_t^1 and H_t^2 for each relay; and execute the NumPyro code to generate the published bandwidth values for all relays. The directories also maintain the full history of the two parameters computed.

As discussed previously, Shadow does not run long enough to produce useful self-reported bandwidth. Hence, we use the actual bandwidth in place of the the self-reported bandwidth in our simulations as we did for the flow-based Python simulations.

ProbFlow **achieves a lower average estimation errors than all the algorithms proposed.** We simulate the 5 different Tor network sampled using each estimation algorithm and starting with zero initial information. All relays are assumed to have equal bandwidth at the start of all the simulations. We calculate the average estimation error of all relays in the 5 simulated network. The results are presented Figure 7a. *ProbFlow* outperforms the rest of the algorithms and maintain a low average estimation error for all relays of less than 20% after only 5 epochs. *MLEFlow* maintain an overall average error between 40% and 50%. Meanwhile, *TorFlow-P* and *sbws** are only able to maintain an error between 60% and 80%. As expected, *MLEFlow* performs best for exit-flagged relays. We plot the average error for exit relays only in Figure 7b. *MLEFlow* was able to maintain an exit relays



(a) All relays error.



(b) Exit relays error.

Fig. 7: Comparison of the average estimation errors of relays in the 5 network samples after 20 epochs.

estimation error of around 20% after 10 epochs. *ProbFlow* still outperforms the other algorithms by reaching an exit error of around 10% after 10 epochs. On the other hand, *TorFlow-P* and *sbws** still have an exit relays estimation error higher than 60%.

ProbFlow **results in considerably narrower error range than other algorithms.** We aggregated the estimation error distribution for all classes of relays, shown in Figure 8. For all classes of relays, *ProbFlow* results in lower average error with a narrow range of error.

Using *ProbFlow* **estimates results in a fairer bandwidth allocation to users on the Tor network.** Finally, compare the impact of using each algorithm estimates on the bandwidth allocated to users in one of the simulated networks. Figure 9 plots the distribution of download speed across the 2000 clients in the 3% network simulated using consensus capacities generated by *ProbFlow*, *MLEFlow*, *TorFlow-P* and *sbws** after 20 epochs of simulation. *ProbFlow* resulted in a more fairly distributed bandwidth between users with a higher average bandwidth of around 146 KB/s. *MLEFlow* estimates resulted in a similar average of users bandwidth at around 143 KB/s. While *TorFlow-P* and *sbws** estimates

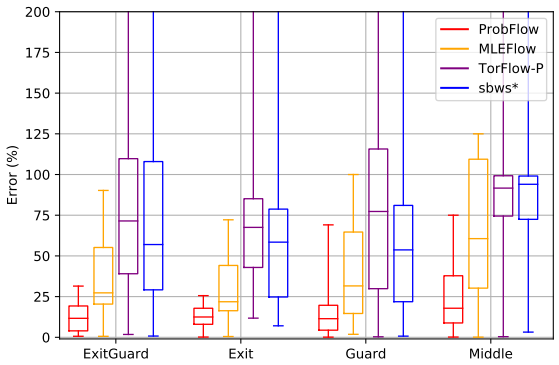


Fig. 8: Relays capacity estimation error distribution using the 4 different algorithms in fully utilized network scenario after 20 epochs computed for the 5 sample of a 3% network.

resulted in an average of around 120 KB/s. Additionally, we can see that the worst performing clients in the network when using *ProbFlow* estimates had a much better performance than when using any other algorithm estimates. In fact, the worst performing 25% of clients had an average performance of 115 KB/s when using *ProbFlow* estimates, while this value was around 80 KB/s for *MLEFlow* and 20 KB/s for *TorFlow-P* and *sbws**. The standard deviation of the bandwidth allocated when using *ProbFlow* is much smaller than when using the other estimated which implies a fairer bandwidth allocation. This can also be depicted in the narrower distribution of bandwidth of *ProbFlow* in Figure 9.

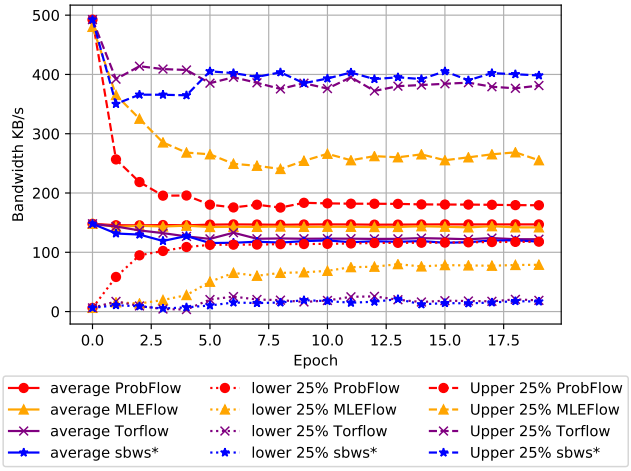


Fig. 9: Aggregate mean progression of client download bandwidth distribution in each consensus epoch in one sample network.

ProbFlow **handles joining relays and relays that change capacity better than other algorithms.** We simulate each algorithm when a new relay joins an already converged network and then changes capacity after some epochs of joining. We consider the case where an exit relay with actual capacity of 15851 KB/s joins the network after 10 epochs of estimation and then changes its actual capacity to 2898 KB/s after another 10 epochs. The results are shown in Figure 10. As can be seen, when the relay first joins, *ProbFlow* estimate converges to the actual capacity in only one epoch with an estimate of 15780 KB/s; while *MLEFlow* takes around 5 epochs to converge to a value of 15700 KB/s; meanwhile, *TorFlow-P* and *sbws** estimates have larger errors throughout the simulation and shows significant fluctuations compared to *ProbFlow* and *MLEFlow*.

When the relay changes its actual capacity after ten epochs to 2898 KB/s, since both *ProbFlow* and *MLEFlow* take into consideration the full history of measurements, their estimates needed some epochs to adjust to the new value. However, *ProbFlow* was able to converge to the new capacity in 5 epochs with a value of 2900 KB/s while *MLEFlow* needed around 15 epochs to converge to 3200 KB/s. The estimates of *TorFlow-P* and *sbws* still showed significant fluctuations.

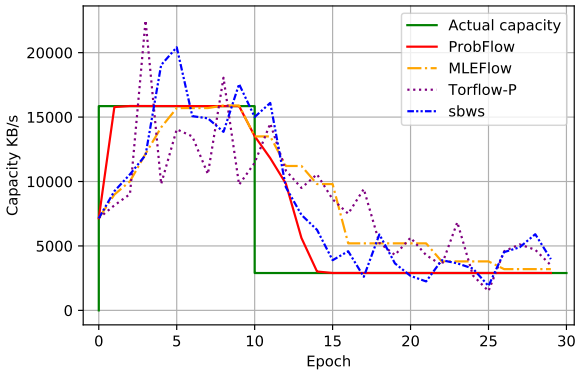


Fig. 10: Exit relay joining after 10 epochs having an actual capacity of 15851 KB/s then changing capacity after 10 epochs to 2898 KB/s.

VII. Security and Privacy Considerations

ProbFlow takes the same inputs as *MLEFlow* and produces the same type of output, except with considerably more accuracy. It therefore inherits most of the security properties of *MLEFlow*, including, importantly, not relying on self-reported bandwidth measurements as are used in the current Tor network. The higher accuracy of *ProbFlow* will result in more uniform throughput for users, enhancing the user experience, which in itself may improve anonymity [8], and potentially reducing Tor's vulnerability to certain throughput-based attacks [19]. One additional consideration that is introduced by *ProbFlow* is the faster convergence of estimates due to bandwidth change. Presently a new relay receives little load for the first few days or weeks of operation as its capacity estimates converge [7], whereas with *ProbFlow* a relay would be able to receive its proper share of traffic nearly immediately. While this improves performance it also makes it easier to quickly introduce rogue relays into the Tor network; as such, mechanisms for detecting such Sybil attacks should be revised [29].

Probabilistic programming also offers interesting opportunities for future security and privacy improvements. For example, while bandwidth measurement data are currently shared publicly (see https://collector.torproject.org/archive/

relay-descriptors/bandwidths/), there is some concern that measurements may be useful in traffic analysis [17]. Using probabilistic programming it would be straightforward to add potential noise to these measurements to improve user privacy and correspondingly adjust the model to account for the noisy data. Alternatively, the model could take into account potential adversarial manipulation of probes by considering that some fraction of measurements could come from a different distribution.

Finally, the programmatic nature of the model enables the estimation algorithm to adapt to change in Tor. AlSabbah and Goldberg survey a large body of work on improving Tor performance and security [1], including changes in path selection, congestion management, etc.; as some of these improvements are deployed, the model can be adjusted to reflect the new functionality of the network much more readily than a manually derived mathematical approximation used in $MLEFlow$.

VIII. Related Work

Many research work focused on improving the performance of the Tor network; an overview can be found in the survey by AlSabah et al. [2]. In this section we summarize the work mainly related to the problem of relay capacity estimation.

Snader and Borisov developed *Eigenspeed* [21] using *opportunistic measurements*. In order to get those measurements, each relay in the network will measure the bandwidth of each other relay it communicates with. Those values are then combined using principal component analysis to compute the estimate of a relay capacity. This algorithm was designed to tackle certain types of misreporting attacks but Johnson et al. [17] showed that it is still vulnerable to other attacks which allow colluding adversaries to inflate their estimated capacity. Johnson et al. [17] thus propose a new mechanism, PeerFlow, to combine the opportunistic measurements from relays that is more robust to inflation attacks. They also derive provable limits on those attacks; however those bounds depend on having a fraction of bandwidth being on trusted nodes. PeerFlow also has slow convergence properties due to the limitations it imposes on changing capacity values.

A new proposal to replace $TorFlow$ uses several probers to measure a relay simultaneously, called *FlashFlow* [27]. The use of several probers simultaneously aims to generate a large network load to max out the capacity of a relay. The mechanism however assumes that a relay capacity is based on a hard limit that can't be exceeded and as the currently deployed algorithms uses traffic that is labeled for bandwidth probing. It should be noted that it is usually easier and cheaper to obtain high peak bandwidth capability than to sustain the same bandwidth for an extended time. In fact, a 10 Gbps servers with traffic restrictions can cost $200 or less while servers with unmetered 10 Gbps traffic cost over $1000 per month. Another requirement of FlashFlow is the fact that it relies on the coordination between several moderate-bandwidth probe servers, while $ProbFlow$ can be deployed with considerably lower capacity probes.

The accuracy of current Tor capacity estimation algorithms was evaluated by Jansen and Johnson [14]. Their simulations showed that there are significant estimation errors as was the case with our simulation results. Their analysis demonstrates that a low self reported bandwidth is a big source of error and will lead to the underestimation of higher capacity nodes. This confirms our claim that that our simulation of $sbws^*$ captures a best case scenario for the algorithm. The results of the paper also motivate the development of capacity estimation methods that don't rely on self-reported bandwidth.

Other approaches aiming to increase load-balancing try to detect and avoid bottlenecks in real time [5], [28], [3], [10]. Other approaches aim to decentralize the bandwidth measurement and operation [11]. Those methods however still rely on relay capacity estimation from the currently deployed algorithms and their contribution could be improved by using $ProbFlow$.

IX. Conclusion

We developed a new relay capacity estimation method, $ProbFlow$, based on a Probabilistic Programming approach. While previous probabilistic model of the network considers one-relay path network, our model takes into account the actual three relays path allocation of the Tor network and is based on the results of bandwidth measurement probes taken over time. The proposed method showed higher estimation accuracy than all proposed algorithms in flow-based and packet-based simulations. This higher accuracy in turn leads to a much better load balancing of user traffic across the network.

Acknowledgment

This material is based upon work supported by C3.ai Digital Transformation Institute for the research award of Securing Critical Cyber-Physical Infrastructure.

References

[1] M. AlSabah and I. Goldberg, "Performance and security improvements for tor: A survey," *ACM Computing Surveys (CSUR)*, vol. 49, no. 2, pp. 1–36, 2016.

[2] ——, "Performance and security improvements for Tor: A survey," *ACM Computing Surveys (CSUR)*, vol. 49, no. 2, p. 32, 2016.

[3] R. Annessi and M. Schmiedecker, "Navigator: Finding faster paths to anonymity," in *2016 IEEE European Symposium on Security and Privacy (EuroS P)*, 2016, pp. 214–226.

[4] K. Bauer, D. McCoy, D. Grunwald, T. Kohno, and D. Sicker, "Low-resource routing attacks against tor," in *Proceedings of the 2007 ACM Workshop on Privacy in Electronic Society*, ser. WPES '07. New York, NY, USA: Association for Computing Machinery, 2007, p. 11–20. [Online]. Available: https://doi.org/10.1145/1314333.1314336

[5] H. Darir, H. Sibai, N. Borisov, G. Dullerud, and S. Mitra, "Tightrope: Towards optimal load-balancing of paths in anonymous networks," in *Proceedings of the 2018 Workshop on Privacy in the Electronic Society*, ser. WPES'18. New York, NY, USA: Association for Computing Machinery, 2018, p. 76–85. [Online]. Available: https://doi.org/10.1145/3267323.3268953

[6] H. Darir, H. Sibai, C.-Y. Cheng, N. Borisov, G. Dullerud, and S. Mitra, "Mleflow: Learning from history to improve load balancing in tor," *Proceedings on Privacy Enhancing Technologies*, vol. 2022, no. 1, pp. 75–104, 2022. [Online]. Available: https://doi.org/10.2478/popets-2022-0005

[7] R. Dingledine, "The lifecycle of a new relay," The Tor Project Blog, https://blog.torproject.org/lifecycle-new-relay, Sep. 2013.

[8] R. Dingledine and N. Mathewson, "Anonymity loves company: Usability and the network effect." in *WEIS*, 2006.

[9] R. Dingledine, N. Mathewson, and P. Syverson, "Tor: The Second-Generation onion router," in *13th USENIX Security Symposium (USENIX Security 04)*. San Diego, CA: USENIX Association, Aug. 2004. [Online]. Available: https://www.usenix.org/conference/13th-usenix-security-symposium/tor-second-generation-onion-router

[10] D. Goulet and M. Perry, "Make relays report when they are overloaded," Tor Proposal 328, https://gitlab.torproject.org/tpo/core/torspec/-/blob/master/proposals/328-relay-overload-report.md, Nov. 2020.

[11] A. Greubel, A. Dmitrienko, and S. Kounev, "Smartor: Smarter tor with smart contracts: Improving resilience of topology distribution in the tor network," in *Proceedings of the 34th Annual Computer Security Applications Conference, ACSAC 2018, San Juan, PR, USA, December 03-07, 2018*. ACM, 2018, pp. 677–691. [Online]. Available: https://doi.org/10.1145/3274694.3274722

[12] R. Jansen, K. Bauer, N. Hopper, and R. Dingledine, "Methodically modeling the tor network," in *5th Workshop on Cyber Security Experimentation and Test (CSET'12)*. Bellevue, WA: USENIX Association, Aug. 2012. [Online]. Available: https://www.usenix.org/conference/cset12/workshop-program/presentation/Jansen

[13] R. Jansen and N. Hopper, "Shadow: Running Tor in a box for accurate and efficient experimentation," in *Proceedings of the 19th Symposium on Network and Distributed System Security (NDSS)*, 2012.

[14] R. Jansen and A. Johnson, "On the accuracy of Tor bandwidth estimation," in *Passive and Active Measurement Conference (PAM)*, 2021.

[15] R. Jansen, J. Tracey, and I. Goldberg, "Once is never enough: Foundations for sound statistical inference in tor network experimentation," in *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021, pp. 3415–3432. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/jansen

[16] R. Jansen, T. Vaidya, and M. Sherr, "Point break: a study of bandwidth denial-of-service attacks against Tor," in *28th USENIX Security Symposium*, 2019, pp. 1823–1840.

[17] A. Johnson, R. Jansen, N. Hopper, A. Segal, and P. Syverson, "PeerFlow: Secure load balancing in Tor," *Proceedings on Privacy Enhancing Technologies*, vol. 2017, no. 2, pp. 74–94, 2017.

[18] juga, "How bandwidth scanners monitor the Tor network," Tor Project Blog, https://blog.torproject.org/aggregation-feed-types/sbws, Apr. 2019.

[19] P. Mittal, A. Khurshid, J. Juen, M. Caesar, and N. Borisov, "Stealthy traffic analysis of low-latency anonymous communication using throughput fingerprinting," in *Proceedings of the 18th ACM conference on Computer and Communications Security*, 2011, pp. 215–226.

[20] M. Perry, "TorFlow: Tor network analysis," in *Proceedings of the 2nd Workshop on Hot Topics in Privacy Enhancing Technologies (HotPETs)*, 2009, pp. 1–14.

[21] R. Snader and N. Borisov, "EigenSpeed: Secure peer-to-peer bandwidth evaluation," in *8th International Workshop on Peer-To-Peer Systems*, R. Rodrigues and K. Ross, Eds. Berkeley, CA, USA: USENIX Association, Apr. 2009.

[22] The Tor Project, "Deploying the simple bandwidth scanner," https://sbws.readthedocs.io/en/latest/DEPLOY.html, 2018.

[23] ——, "Tor directory protocol, version 3," https://gitweb.torproject.org/torspec.git/tree/dir-spec.txt, 2020.

[24] ——, "Tor metrics: Servers," https://metrics.torproject.org/networksize.html, 2020.

[25] ——, "Tor metrics: Users," https://metrics.torproject.org/userstats-relay-country.html, 2020.

[26] F. Thill, "Hidden service tracking detection and bandwidth cheating in Tor anonymity network," Ph.D. dissertation, University of Luxembourg, 2014.

[27] M. Traudt, R. Jansen, and A. Johnson, "Flashflow: A secure speed test for tor," 2020.

[28] T. Wang, K. Bauer, C. Forero, and I. Goldberg, "Congestion-aware path selection for Tor," in *International Conference on Financial Cryptography and Data Security*, 2012, pp. 98–113.

[29] P. Winter, R. Ensafi, K. Loesing, and N. Feamster, "Identifying and characterizing sybils in the tor network," in *25th USENIX Security Symposium (USENIX Security 16)*, 2016, pp. 1169–1185.

[30] M. K. Wright, M. Adler, B. N. Levine, and C. Shields, "The predecessor attack: An analysis of a threat to anonymous communications systems," *ACM Transactions on Information and System Security (TISSEC)*, vol. 7, no. 4, pp. 489–522, 2004.

APPENDIX

In this appendix, we present the code used to implement the *probabilistic programs* presented in this paper. The code used to implement the model of $MLEFlow$, i.e. Algorithm 2 is presented in Figure 11. We start the code by specifying the prior distribution through *scale0* and *shape0*. As can be seen *shape0* was set to 1 since a Weibull distribution with a shape parameter of 1 is equivalent to an exponential distribution. The capacities are then sampled using the exponential distribution. The next step is implementing the probabilistic model of $MLEFlow$ for all the measurements using the *numpyro.plate* command. We use the *DoubleAffineTransform* to implement the probabilistic model $\frac{C^*}{X+1}$. The guide, presented in Algorithm 1, is implemented using the code of Figure 12. The guide parametrize the shape and scale of the Weibull distribution and samples the estimated capacities.

To be able to run the model and guide for a specific relay, the commands in Figure 13 are run in *Pyro*.

The implementation of $ProbFlow$ model, Algorithm 3 is presented in Figure 14. It should be noted that the inputs *sum bottleneck* refers to parameter H_t^2, while the input *sum weight* refers to $1 - H_t^1$. The parameters H_t^1 and H_t^2 are computed using the weights of the classes, the measurements of each relay and the weight of the relay. The implementation of the computation of those parameters is the major difference between the models of Algorithm 2 and Algorithm 3. The other difference is the arguments of the *DoubleAffineTransform* transformation. Similarly, we start the code of Figure 14, by specifying the prior as an exponential distribution. We then use the *numpyro.plate* and the *DoubleAffineTransform* to implement the probabilistic model derived for $ProbFlow$, i.e. $\frac{C^*-XH^2}{XH^1+1}$ for each measurement of the relay.

We should also note the transformation that we are using in both the models, *DoubleAffineTransform*. This transformation maps a random variable X to $\frac{loc_1+scale_1 X}{loc_2+scale_2 X}$. This transformation is not yet supported in *numpyro*, hence we wrote the code of this transformation manually.

```
def model(guess_scale,  number_of_paths, used_weight_guard, used_weight_middle,  used_weight_exit, data_of_relay):
    scale0 = guess_scale
    shape0 = 1
    C=numpyro.sample("capacity", dist.Weibull(scale0, shape0))
    with numpyro.plate("obs", (data_of_relay).size) as i:
        transform2 = [DoubleAffineTransform(loc1=C, scale1=0, loc2=1, scale2=1, total_nb_paths=number_of_paths, obs=data_of_relay[i] )]
        numpyro.sample("obs_{}".format(i), dist.TransformedDistribution(dist.Poisson(number_of_paths*(used_weight_guard[i]+used_weight_middle[i]+used_weight_exit[i])),
                                                                        transform2), obs=data_of_relay[i])
```

Fig. 11: Algorithm 2 implementation in *numpyro*.

```
def guide(guess_scale,  number_of_paths, used_weight_guard, used_weight_middle,  used_weight_exit, data_of_relay):
    scale_q = numpyro.param("scale_q", guess_scale, constraint=constraints.positive)
    shape_q = numpyro.param("shape_q", 1, constraint=constraints.positive)
    numpyro.sample("capacity", dist.Weibull(scale_q, shape_q))
```

Fig. 12: Algorithm 1 implementation in *numpyro*.

```
svi = SVI(model, guide, optim=Adam(0.01), loss=Trace_ELBO())
svi_result= svi.run(jax.random.PRNGKey(0),num_steps_inference, scale_estimate[i], number_of_paths, weight_used_in_generation_guard[i],
                                                                weight_used_in_generation_middle[i], weight_used_in_generation_exit[i],observations[i])
```

Fig. 13: Running the Probabilistic program in *numpyro*.

```
def model(guess_scale,  sum_bottleneck, sum_weight, number_of_paths, used_weight_guard, used_weight_middle,  used_weight_exit, data_of_relay):
    scale0 = guess_scale
    shape0 = 1
    C=numpyro.sample("capacity", dist.Weibull(scale0, shape0))
    with numpyro.plate("obs", (data_of_relay).size) as i:
        transform2 = [DoubleAffineTransform(loc1=C, scale1=-sum_bottleneck, loc2=1, scale2=1-sum_weight[i], total_nb_paths=number_of_paths, obs=data_of_relay[i] )]
        numpyro.sample("obs_{}".format(i), dist.TransformedDistribution(dist.Poisson(number_of_paths*(used_weight_guard[i]+used_weight_middle[i]+used_weight_exit[i])),
                                                                        transform2), obs=data_of_relay[i])
```

Fig. 14: Algorithm 3 implementation in *numpyro*.