ABRA CADABRA: Magically Increasing Network Utilization in Tor by Avoiding Bottlenecks

John Geddes
University of Minnesota
geddes@cs.umn.edu

Mike Schliep
University of Minnesota
schliep@cs.umn.edu

Nicholas Hopper
University of Minnesota
hopper@cs.umn.edu

ABSTRACT

Like many routing protocols, the Tor anonymity network has decentralized path selection, in that clients locally and independently choose paths. As a result, network resources may be left idle, leaving the system in a suboptimal state. This is referred to as the price of anarchy, where agents acting in an uncoordinated fashion make poor decisions when viewed in a global context. In this paper we introduce ABRA, the avoiding bottleneck relay algorithm, which can be used to allow some coordination between clients and relays with the aim of increasing network utilization. At peak performance, using ABRA for circuit selection results in almost 20% higher network bandwidth usage compared to vanilla Tor. We find that ABRA significantly outperforms previously suggested circuit selection algorithms based on latency and congestion, and attains similar throughput to a fully centralized online algorithm, while an offline algorithm with knowledge of future requests could achieve up to 80% more network utilization than vanilla Tor. Finally, we perform a privacy analysis of ABRA against a passive and active adversary trying to reduce anonymity of clients and increase their view of the Tor network. We find that the algorithm does not enable new passive attacks and that colluding relays experience a minor increase in the fraction of streams compromised when acting in an adversarial manner.

1. INTRODUCTION

Tor [7] is one of the most popular and widely used low-latency anonymity systems, with 2 million daily clients supported by a network of over 6000 volunteer relays. Tor clients tunnel traffic through a sequence of 3 relays called a *circuit*. Traffic sent over the circuit is encrypted so that no single relay can learn the identity of both the client and destination. With a limited amount of resources available from the volunteer relays, it is important to distribute traffic to utilize these resources as efficiently as possible. Doing so ensures high performance to latency sensitive applications and attracts potential clients to use the network, increasing anonymity for all users.

When a Tor client first starts, it fetches a list of all relays and their information from a directory authority. The client then starts building circuits, selecting relays at random weighted by the relay's bandwidth. Tor clients attempt to keep a pool of roughly 10 open circuits available at all times. When a new download starts, the Tor client creates a stream and selects a circuit which is then used for all new streams for the following 10 minutes. After this time period a new circuit is selected, with the old circuit being destroyed after all streams have finished using it.

In routing, when network users make decisions locally in accordance to their own interest, the problem of *selfish routing* [28, 29] can occur. In this case the network as a whole is left in a sub-optimal state, with network resources underutilized, degrading client performance. While Tor clients are not necessarily selfish, circuit selections are made locally without consideration of the state of the global network. To this end, several researchers have investigated improvements to path selection [35], such as to consider latency [2] or relay congestion [36] in an attempt to improve client performance.

In this paper we introduce ABRA, the avoiding bottleneck relays algorithm, which can be used for circuit selection that intends to maximize network utilization. By borrowing techniques from delay-weighted capacity (DWC) routing [37] we design a system in which relays can gossip information to clients that allows client to select circuits for new streams more intelligently. By avoiding bottleneck relays with especially low available bandwidth when selecting circuits, network resources are much more efficiently utilized, leading to both higher total network bandwidth and faster client downloads.

We make the following major contributions:

- We design and implement an algorithm which can be used by relays to estimate which circuits they are bottlenecks on, allowing them to calculate a weight to gossip to clients. Similar to DWC routing, this allows clients to avoid bottlenecks when selecting circuits to use for new streams.
- To explore optimal network utilization we implement a central authority (CA) that makes circuit selection decisions for all clients. We design an algorithm the CA can use with its global view of the network that allows more accurate identification of bottlenecks, allowing the central authority to route clients around bottlenecks in real time.
- Additionally, we design a genetic algorithm that is able

WPES '16, October 24, 2016, Vienna, Austria.

ISBN 978-1-4503-4569-9/16/10...$15.00

DOI: http://dx.doi.org/10.1145/2994620.2994630

to precompute circuit selection offline for a fixed set of download times. This allows us to perform competitive analysis against the online algorithm, acting as a lower bound on optimal network utilization.
- All algorithms are analyzed in a large scale simulated environment using the Shadow simulator [15]. The ABRA algorithm is compared against other path selection strategies that consider congestion and latency when picking circuits.
- Finally, we perform privacy analysis when using ABRA for circuit selection. We consider both information that can be learned from a passive adversary, and examine how an active adversary could attempt to abuse the algorithm to increase the fraction of streams they are able to compromise.

2. BACKGROUND

In this section we discuss some of the Tor architecture, basics on how clients operate and send data through the network, and related work on increasing performance in Tor.

2.1 Tor Architecture

When a client first joins the Tor network, it downloads a list of relays with their relevant information from a directory authority. The client then creates 10 *circuits*, each passing through a guard, middle, and exit relay. The relays used for each circuit are selected at random weighted by bandwidth. For each TCP connection the client wishes to make, Tor creates a *stream* which is then assigned to a circuit; each circuit will typically handle multiple streams concurrently. The Tor client will find a viable circuit that can be used for the stream, which will be then be used for the next 10 minutes or until the circuit becomes unusable. After the 10 minute window a new circuit is selected to be used for the following 10 minutes, with the old circuit being destroyed after any stream still using it has ended.

Internal to the overlay network, relay pairs establish a single TLS connection for *all* communication between them. To send traffic through a circuit, data is packed into 512-byte onion-encrypted cells using secret keys negotiated with each relay during the circuit building process. Once a relay has received a cell, it peels off its layer of encryption, finds the next relay on the circuit to send to, and places it on a circuit queue where it waits to be sent. After a cell has traversed the entire circuit, the exit recovers the initial data sent by the client, which is forwarded to the end destination.

2.2 Related Work

One of the major keys to increasing anonymity for Tor users is to ensure a large anonymity set, that is, a large user base. To do so Tor needs to offer low latency to its clients; bad performance in the form of slow web browsing can lead to fewer users using the system overall. To this end, there has been a plethora of research looking to address ways to increase performance in Tor. These roughly fall into 4 areas: scheduling, selection, transports, and incentives.

Scheduling: Internally Tor is constantly making scheduling decisions on what queue to service and how much should be processed. These decisions happen on all levels, between streams, circuits, and connections. On the circuit level, researches have proposed methods to classify latency sensitive circuits, either prioritizing them [3,34] or outright throttling noisy ones [18]. At the stream level, AlSabah *et al.* [4] investigated alternative mechanisms for flow control. And at the connection level, the KIST algorithm [14] was proposed as a method to improve the interaction of Tor's scheduling with the kernel.

Relay Selection: When creating circuits, Tor selects relays at random weighted by their bandwidth. Determining the bandwidth is a non-trivial issue, and several papers [32, 33] have examined a range of methods, from using self-reported values, central nodes making direct measurements, and peer-to-peer methods relying on existing relays. There also has been a lot of research in improving relay selection [2, 6, 12, 30, 31, 35, 36] for circuit creation. The methods proposed include incorporating latency and congestion measurements, using a virtual coordinate system to estimate latencies, and adjusting the weighting strategy, all in an attempt improve overall client performance.

Transport: One of the noted performance issues in Tor is the fact that the single TLS connection between relays can cause unnecessary blocking, where circuits could keep sending data but TCP mechanisms prevent it [8]. Reardon [26] attempted to address this by implementing TCP-over-DTLS allowing a connection to be dedicated to each circuit. In a similar vein, several papers [5,9,10] have proposed increasing the number of TCP connections between relays and scheduling circuits between them in an attempt to avoid unnecessary blocking. Nowlan *et al.* also introduced uTCP and uTLS [23, 24], which allows for out-of-order delivery in Tor so it can process cells from circuits that are not blocking.

Incentives: While the previous lines of research involved improving efficiency in how Tor handles traffic, another set looked at potential ways to incentivize clients to also contribute bandwidth as a relay, increasing the overall network resources available to clients. This was first explored by Ngan, Dingledine, and Wallach [22], who suggested prioritizing traffic from relays providing high quality service in the Tor network. Jansen, Hopper, and Kim [16] extend this idea, allowing relays to earn credits which can be redeemed for more prioritized traffic. Building on this, Jansen, Johnson, and Syverson [17] introduce a more lightweight solution that allows for the same kind of prioritization of traffic without as much overhead.

3. ABRA FOR CIRCUIT SELECTION

The avoiding bottleneck relays algorithm is based on the delay-weighted capacity (DWC) routing algorithm[1], designed with the goal of minimizing the "rejection rate" of requests in which the bandwidth requirements of a request cannot be satisfied. To accomplish this the algorithm must utilize network resources as efficiently as possible when performing path selection for requests, making it a good model for a Tor circuit selection algorithm that attempts to best utilize relay bandwidth. The main difference between the algorithms is that in DWC routing an entity with a global view of the network is used to identify bottlenecks and route requests around them, which is infeasible for Tor.

Instead, in ABRA relays themselves estimate on which circuits they are bottlenecks. With this information they can then compute their own DWC weight and gossip the weight value periodically to all clients with a circuit through them. To calculate the DWC weight for a link the algorithm needs to know how many paths the link is a bottleneck on. To

[1]Details of the DWC algorithm can be found in Appendix A

Figure 1: Example of circuit bandwidth estimate algorithm. Each step the relay with the lowest bandwidth is selected, its bandwidth evenly distributed among any remaining circuit, with each relay on that circuit having their bandwidth decremented by the circuit bandwidth.

locally identify bottleneck circuits, we make three observations: (1) to be a bottleneck on *any* circuit the relay's bandwidth should be fully consumed (2) all bottleneck circuits should be sending more cells than non-bottleneck circuits, and (3) the number of cells being sent on bottleneck circuits should be fairly uniform. The last point is due to flow control built into Tor, where the sending rate on a circuit should converge to the capacity of the bottleneck relay.

All of this requires measuring the bandwidth used by a circuit, and needs to be done in a way that does not *over* estimate the bandwidth of bulky circuits or *under* estimate quiet but bursty circuits. To accomplish this we use an algorithm with two parameters. First is the bandwidth *window*, in which we keep track of all traffic sent on a circuit for the past w seconds. The other parameter is the bandwidth *granularity*, a sliding window of g seconds that we scan over the w second bandwidth window to find the maximum number of cells transferred during any g second period. That maximum number of cells divided by g seconds is then assigned as the circuit bandwidth. (So if we assign the circuit a bandwidth of b cells per second then in the last w seconds, the circuit transferred at most $g \times b$ cells in any g-second interval)

With circuits assigned a bandwidth value we need a method to decide whether a relay is a bottleneck on a given circuit. Notice that if circuit C is using much less bandwidth than other circuits utilizing relay R, then R cannot be the bottleneck on C So in an idealized setting, R would have several non-bottleneck circuits that use very few cells per second of bandwidth, and m bottleneck circuits that each use about $1/m$ of R's bandwidth. Due to noise in the measurements, we expect that the measured bandwidth of both the bottleneck and non-bottleneck circuits could vary, so we attempt to identify this set of circuits by clustering circuits into bottleneck and non-bottleneck groups based on their measured bandwidths.

For this we consider several clustering algorithms. In the Jenks natural breaks algorithms, we group data into n classes that minimize the in-class variance. We can either try to directly group circuits into $n = 2$ classes (*Jenks Two*), or keep increasing n until the goodness of variance fit crosses a threshold τ (*Jenks Best*). The circuits that then fall into the 2^{nd} or n^{th} class are labeled as bottlenecks. A third method we consider is the *head/tail* clustering algorithm which is useful when the underlying data has a long tail. The data is split into the head or tail class, with all values greater than the mean assigned to the head. If the fraction of data that ends up in the head class is less than some threshold, the process is repeated using the head class as the new data set. Finally, we also consider a multimodal gaussian *kernel density* estimator, which is explained in greater detail in the Appendix.

After relays have identified the circuits they are bottlenecks on, they compute their DWC weight as the sum of the inverse of those circuits' bandwidths, $weight = \sum \frac{1}{bw_i}$. This weight is then periodically gossiped to all clients that have a circuit through the relay. In the centralized DWC algorithm paths are always selected with the lowest combined DWC weight across all nodes in the path. However since there is a delay between when circuits become bottlenecks and notify clients we want to avoid clients over-burdening relays that might be congested. Instead clients will select circuits at random based on the DWC weight information. First the client checks if there are any circuits with DWC weight of 0, indicating that no relay on the circuit is a bottleneck on *any* circuit. If there are any, all zero weight circuits are added to a list which the client selects from randomly weighted by the circuit bandwidth, which is the lowest advertised bandwidth of each relay in the circuit. If there are no zero-weight circuits, the client then defaults to selecting amongst the circuits randomly weighted by the *inverse* of the DWC weight, since we want to bias our selection to circuits with lower weights.

4. MAXIMIZING NETWORK USAGE

In addition to analyzing improvements to network utilization compared to vanilla Tor, we are also interested in how close using ABRA for circuit selection is to an optimal solution. To this end we design and implement online and offline algorithms with the sole purpose of achieving maximal network utilization. The online algorithm is confined to processing requests as they are made, meaning that at any one time it can know all current active downloads and which circuits they are using, but has no knowledge of *how long* the download will last or the start times of any future downloads. The offline algorithm, on the other hand, has full knowledge of the entire set of requests that will be made, both the times that the downloads will start and how long they will last.

4.1 Online Algorithm

While an actual centralized authority making all circuit selection decisions is obviously impractical due to privacy concerns, it allows us to see exactly how much spare capacity there exists in the network and how close ABRA performs to this ideal scenario. For the online algorithm we use the same basic idea as ABRA: identify which relays are bottlenecks on active circuits, calculate their weight based on the bandwidth of circuits they are bottlenecks on, and then select circuits for downloads with the lowest weight. Since the

central authority has a global view of the network it can, in real time, make these calculations and direct clients around potential bottlenecks.

For this we need an algorithm to identify bottleneck relays given a set of active circuits. The algorithm has a list of relays and their bandwidth, along with the list of active circuits. First we extract the relay with the lowest bandwidth per circuit, defined as $r_{circBW} = r_{bw}/|\{c \in circuits | r \in c\}|$. This relay is the bottleneck on every circuit it is on in the circuit list, so we can assign it a weight of $r_w = 1/r_{bw}$. Afterwards, for each circuit that r appears on we can assign the circuit a bandwidth value of r_{circBW}. Additionally, for every other relay r' that appears in those circuits, we decrement their available bandwidth $r'_{bw} = r'_{bw} - r_{circBW}$, accounting for the bandwidth being consumed by the circuit. Once this is completed all circuits that r appears on are removed from the circuit list. Additionally, any relay that has a remaining bandwidth of 0 (which will always include r) is removed from the list of relays[2]. This is repeated until there are no remaining circuits in the circuit list[3]. An example of this process is shown in Figure 1.

Now when a download begins, the algorithm knows the set of circuits the download can use and a list of all currently active circuits. The algorithm to compute relay weights is run and the circuit with the lowest combined weight is selected. If there are multiple circuits with the same lowest weight then the central authority selects the circuit with the highest bandwidth, defined as the minimum remaining bandwidth (after the weight computation algorithm has completed) across all relays in the circuit.

4.2 Offline Algorithm

An offline algorithm is allowed access to the *entire* input set while operating. For circuit selection in Tor this means prior knowledge of the start and end time of every download that will occur. While obviously impossible to do on a live network, even most Tor client models [13] consist of *client* start times, along with file sizes to download and how long to pause between downloads. To produce a comparison, we use the offline algorithm to rerun circuit selection on observed download start and end times. We then rerun the downloads with the same exact start and end times with the new circuit selection produced by the offline algorithm to see if network utilization improves. As input the algorithm takes a list of downloads d_i with their corresponding start and end times (s_i, e_i) and set of circuits $C_i = \{c_{i_1}, c_{i_2}, ...\}$ available to the download, along with the list of relays r_j and their bandwidth bw_j. The algorithm then outputs a mapping $d_i \rightarrow c_{i_j} \in C_i$ with $c_{i_j} = \langle r_1, r_2, r_3 \rangle$.

While typical routing problems are solved in terms of graph theory algorithms (e.g. max-flow min-cut), the offline circuit selection problem more closely resembles typical job scheduling problems [11]. There are some complications when it comes to precisely defining the machine environment, but the biggest issues is the scheduling problems that most closely resemble circuit selection have been discovered to be NP-hard. Due to these complications we develop a genetic algorithm, which have been shown to perform well on other job scheduling problems, in order to compute a lower-bound on the performance of an optimal offline solution.

[2]A proof showing any such relay will no longer be on any remaining circuit is given in Appendix C

[3]Pseudocode for this is shown in Algorithm 2 in Appendix D.

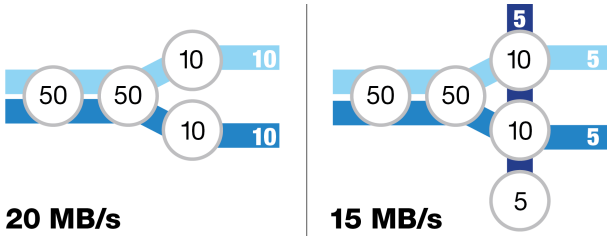


Figure 2: Example showing that adding an active circuit resulted in network usage dropping from 20 to 15 MBps

.

The important pieces of a genetic algorithm are the seed solutions, how to breed solutions, how to mutate a solution, and the fitness function used to score and rank solutions. In our case a solution consists of a complete mapping of downloads to circuits. To breed two parent solutions, for each download we randomly select a parent solution, and use the circuit selected for the download in that solution as the circuit for the child solution. For mutation, after a circuit has been selected for the child solution, with a small probability $m\%$ we randomly select a relay on the circuit to be replaced with a different random relay. Finally, to score a solution we want a fitness function that estimates the total network utilization for a given circuit to download mapping.

To estimate network utilization for a set of downloads, we first need a method for calculating the bandwidth being used for a fixed set of active circuits. For this we can use the weight calculation algorithm from the previous section to estimate each active circuit's bandwidth, which can then be aggregated into total network bandwidth. To compute the entire amount of bandwidth used across an entire set of downloads, we extract all start and end times from the downloads and sort them in ascending order. We then iterate over the times, and at each time t_i we calculate the bandwidth being used as bw_i and then increment total bandwidth $totalBW = totalBW + bw_i \cdot (t_{i+1} - t_i)$. The idea here is that between t_i and t_{i+1}, the same set of circuits will be active consuming bw_i bandwidth, so we add the product to total bandwidth usage. This is used for the fitness function for the genetic algorithm, scoring a solution based on the estimated total bandwidth usage. We consider two different methods for seeding the population, explained in the next section.

4.3 Circuit Sets

In both the online and offline circuit selection algorithms, each download has a set of potential circuits that can be selected from. For the online algorithm we want to use the full set of all valid circuits. We consider a circuit r_1, r_2, r_3 valid if r_3 is an actual exit relay, and if $r_1 \neq r_2$, $r_2 \neq r_3$, and $r_1 \neq r_3$. Note we do not require that r_1 actually has the guard flag set. This is because while there are mechanisms internally in Tor that prevent any use of a non-exit relay being used as an exit relay, there is nothing stopping a client from using a non-guard relay as their guard in a circuit. We also remove any "duplicate" circuits that contain the same set of relays, just in a different order. This is done since neither algorithm considers inter-relay latency, only relay bandwidth, so there is no need to consider circuits with identical relay sets.

For the offline algorithm we need to carefully craft the set of circuits as to prevent the genetic algorithm from getting

Bandwidth Granularity	100 ms				1 second			
Bandwidth Window	1s	2s	5s	10s	1s	2s	5s	10s
Jenks Two	1.67	1.79	3.56	8.69	1.80	2.39	5.28	13.25
Jenks Best	1.11	1.14	1.91	3.87	1.00	1.18	2.25	4.70
Head/Tail	0.74	0.92	1.66	4.36	0.90	1.13	2.45	6.14
KernelDensity	0.81	0.82	0.85	0.93	0.79	0.79	0.95	1.62

Table 1: The bottleneck clustering methods mean square error across varying bandwidth granularity and window parameters, with red values indicating scores less than weighted random estimator.

stuck at a local maximum. The motivation behind generating this pruned set can be seen in Figure 2. It shows that we can add an active circuit and actually *reduce* the amount of bandwidth being pushed through the Tor network. In the example shown it will always be better to use one of the already active circuits shown on the left instead of the new circuit seen on the right side of the graphic. There are two main ideas behind building the pruned circuit set: (1) when building circuits always use non-exit relays for the guard and middle if possible and (2) select relays with the highest bandwidth. To build a circuit to add to the pruned set, the algorithm first finds the exit relay with the highest bandwidth. If none exist, the algorithm can stop as no new circuits can be built. Once it has an exit, the algorithm then searches for the non-exit relay with the highest bandwidth, and this relay is used for the middle relay in the circuit. If one is not found it searches the remaining exit relays for the highest bandwidth relay to use for the middle in the circuit. If it still cannot find a relay, the algorithm stops as there are not enough relays left to build another circuit. The search process for the middle is replicated for the guard, again stopping if it cannot find a suitable relay. Now that the circuit has a guard, middle, and exit relay, the circuit is added to the pruned circuit set and the algorithm calculates the circuit bandwidth as the minimum bandwidth across all relays in the circuit. Each relay then decrements its relay bandwidth by the circuit bandwidth. Any relay that now has a bandwidth of 0 is permanently removed from the relay list. This is repeated until the algorithm can no longer build valid circuits[4].

Using these sets allows us to explore the full potential performances that can be realized when considering both relay and circuit selection. To analyze the impacts on circuit selection alone, we also run the algorithms using the original circuit set. For each download, we extract the original circuits available in the vanilla experiments. Then when selecting a circuit for the download in the online and offline algorithm we only pick from those original circuits.

5. EXPERIMENTAL SETUP

In this section we discuss our experimental setup and some implementation details of the algorithms used.

5.1 Shadow

To empirically test how our algorithms would perform in the Tor network, we use Shadow [1, 15], a discrete event network simulator with the capability to run actual Tor code in a simulated network environment. Shadow allows us to set up a large scale network configuration of clients, relays, and servers, which can all be simulated on a single machine. This lets us run experiments privately without operating on the actual Tor network, avoiding potential privacy concerns of dealing with real users. Additionally, it lets us have a global view and precise control over every aspect of the network, which would be impossible in the live Tor network. Most importantly, Shadow performs deterministic runs, allowing for reproducible results and letting us isolate exactly what we want to test for performance effects.

Our experiments are configured to use Shadow v1.9.2 and Tor v0.2.5.10. We use the large network configuration distributed with Shadow which consists of 500 relays, 1350 web clients, 150 bulk clients, 300 performance clients, and 500 servers. The default client model in Shadow [13–15] downloads a file of a specific size, and after the download is finished it chooses how long to pause until it starts the next download. Web clients download 320 KB files and randomly pause between 1 and 60,000 milliseconds before starting the next download; bulk clients download 5 MB files with no break between downloads; and performance clients are split into three groups, downloading 50 KB, 1 MB, and 5 MB files, respectively, with 1 minute pauses between downloads. Additionally, since the offline algorithm uses deterministic download times, we introduce a new fixed download model. In this model each client has a list of downloads it will perform with each download containing a start time, end time, and optional circuit to use for the download. Instead of downloading a file of fixed size it downloads non-stop until the end time is reached. In order to generate a fixed download experiment, we use the download start and end times seen during an experiment running vanilla Tor using the default client model.

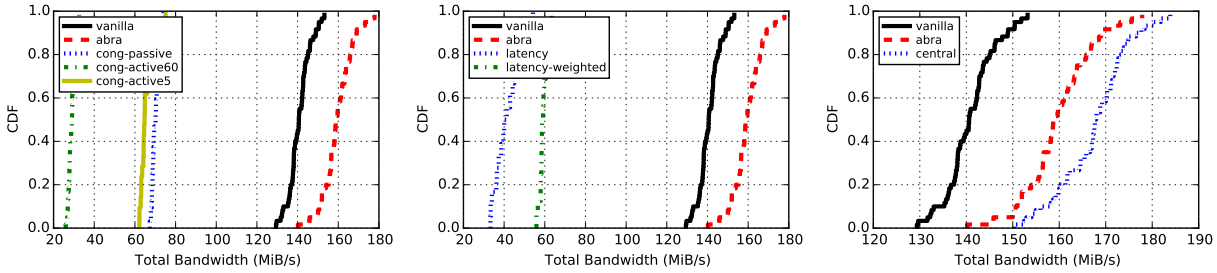
To measure network utilization, every 10 seconds each relay logs the total number of bytes it has sent during the previous 10 second window. This allows us to calculate the total relay bandwidth being used during the experimental run, with higher total bandwidth values indicating better utilization of network resources. When using the default client model we also examine client performance, with clients reporting the time it took them to download the first byte, and time to download the entire file.

5.2 Implementations

Avoiding Bottleneck Relays Algorithm: To incorporate ABRA in Tor we first implemented the local weight calculation method described in Section 3. Each circuit keeps track of the number of cells received over each 100 millisecond interval for the past w seconds. We created a new `GOSSIP` cell type which relays send with their DWC weight downstream on each circuit they have. To prevent gossip cells from causing congestion, relays send the gossip cells on circuits every 5 seconds based on the circuit ID; specifically when $now() \equiv circID \mod 5$. For clients we modified the `circuit_get_best` function to select circuits randomly based on the sum of relay weights as discussed in Section 3.

Online and Offline Algorithms: When using the online

[4]See Algorithm 4 in Appendix D for pseudocode.



(a) Congestion Aware **(b)** Latency Aware **(c)** Central Authority

Figure 3: Total network bandwidth utilization when using vanilla Tor, ABRA for circuit selection, the congestion and latency aware algorithms, and the central authority.

and offline algorithm with the fixed download client model we create a tool that precomputes circuit selection using as input the list of downloads, with their respective start and stop times, along with which set of circuits described in Section 4.3 to consider for the downloads. The computed mapping of downloads to circuits is used by Shadow to manage which circuits clients need to build and maintain, along with which circuit is selected each time a download starts. For the offline algorithm the genetic algorithm was run for 100 rounds with a breed percentile of $b = 0.2$ and elite percentile $e = 0.1$. This was done for each circuit set, with mutation set to $m = 0$ for the pruned and original circuit sets, and $m = 0.01$ when using the full circuit set. After 100 rounds the population with the highest score saved its circuit selection for every download.

While the offline algorithm can only operate in the fixed download model, the online algorithm can also work in the default client mode. For this we introduce a new central authority (CA) node which controls circuit selection for every client. Using the Tor control protocol the CA listens on the `CIRC` and `STREAM` events to know what circuits are available at each client and when downloads start and end. Every time a `STREAM NEW` event is received it runs the algorithm described in Section 4.1 over the set of active circuits. This results in each relay having an assigned DWC weight and the CA simply picks the circuit on the client with the lowest combined weight.

Congestion and Latency Aware: Previous work has been done on using congestion [36] and latency [2] to guide circuit selection. For congestion aware path selection we use the implementation described in [35]. Clients get measurements on circuit round-trip times from three sources: (1) time it takes to send the last `EXTEND` cell when creating the circuit (2) a special `PROBE` cell sent right after the circuit has been created, and (3) the time it takes to create a connection when using a circuit. For each circuit the client keeps track of the minimum RTT seen rtt_{min}, and whenever it receives an RTT measurement calculates the congestion time $t_c = rtt - rtt_{min}$. When a client selects a circuit it randomly chooses 3 available circuits and picks the one with the lowest congestion time. Additionally we add an active mode where `PROBE` cells are sent every n seconds, allowing for more up to date measurements on *all* circuits the client has available. Instead of using network coordinates to estimate latency between relays as done in [2], we create a latency aware selection algorithm that uses directly measured RTT times via `PROBE` cell measurements. With these measurements the client either selects the circuit with the lowest rtt_{min}, or selects amongst available circuits randomly weighted by each circuit's rtt_{min}.

5.3 Consistency

To keep the experiments as identical as possible, every experiment was configured with a central authority and had relays gossiping their weight to clients. If the experiment was testing a local circuit selection algorithm, the central authority would return a circuit ID of 0 which indicates that the client itself should select a circuit. At this point the client would use either the vanilla, congestion aware, latency aware, or ABRA algorithms to select a circuit. This way any latency or bandwidth overhead incurred from the various algorithms is present in every experiment, and differences between experimental results are all due to different circuit selection strategies.

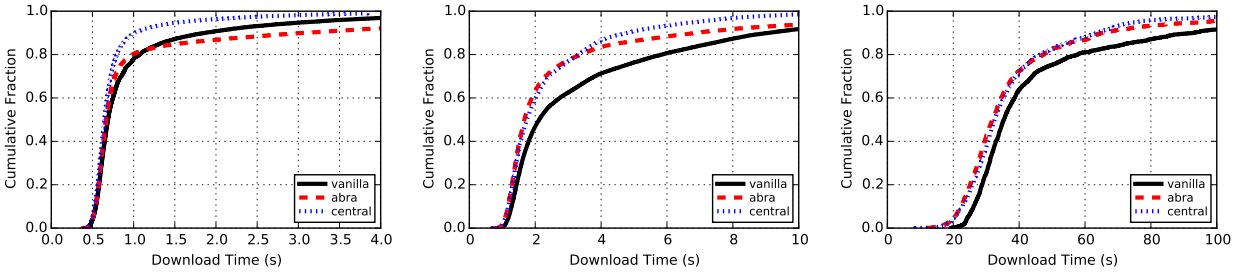
6. PERFORMANCE

In this section we compare the performance of using ABRA for circuit selection to vanilla Tor, congestion-based and latency-aware circuit selection, and our centralized algorithms.

6.1 ABRA Parameters

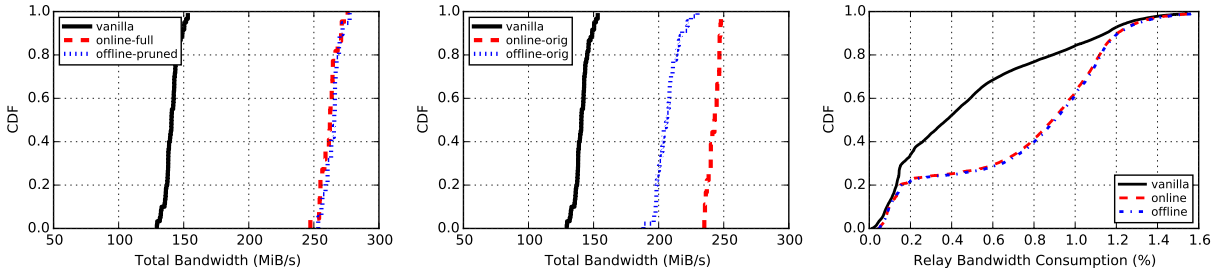
With the various parameters and clustering methods available to the algorithm, the first thing we are interested in is which parameters result in the most accurate bottleneck estimation. To measure this we configured an experiment to run with the central authority described in Section 5.2 making circuit selections. Every time the central authority selects a circuit for a client, for every relay it outputs how many circuits the relay is a bottleneck on. Additionally, during the run every relay periodically outputs the entire 10 second bandwidth history of every circuit it is on. With this information we can compute offline which circuits every relay would have classified itself as a bottleneck on, depending on the parameters used. For comparing estimates, every time the central authority outputs its bottleneck figures, every relay that outputs their circuit history within 10 simulated milliseconds will have their local bottleneck estimate compared to the estimate produced by the central authority.

Table 1 shows the mean-squared error between the bottleneck estimations produced by the central authority and local estimates with varying parameters and clustering methods. The bandwidth granularity was set to either 100 millisec-



(a) Time to First Byte **(b)** Web Download Times **(c)** Bulk Download Times

Figure 4: Download times and client bandwidth compared across circuit selection in vanilla Tor, using ABRA, and the centralized authority.



(a) Full and Pruned Circuits **(b)** Original Circuits **(c)** Relay Capacity

Figure 5: Total network bandwidth when using the online and offline circuit selection algorithms while changing the set of available circuits, along with the relay bandwidth capacity of the best results for both algorithms.

onds or 1 second, and the bandwidth window was picked to be either 1, 2, 5, or 10 seconds. Every value in black performed *worse* than a weighted random estimator, which simply picked a random value from the set of all estimates computed by the central authority. While the kernel-density estimator produced the lowest mean-squared error *on average*, the lowest error value came from when we used the head/tail clustering algorithm with bandwidth granularity set to 100 milliseconds bandwidth window at 1 second.

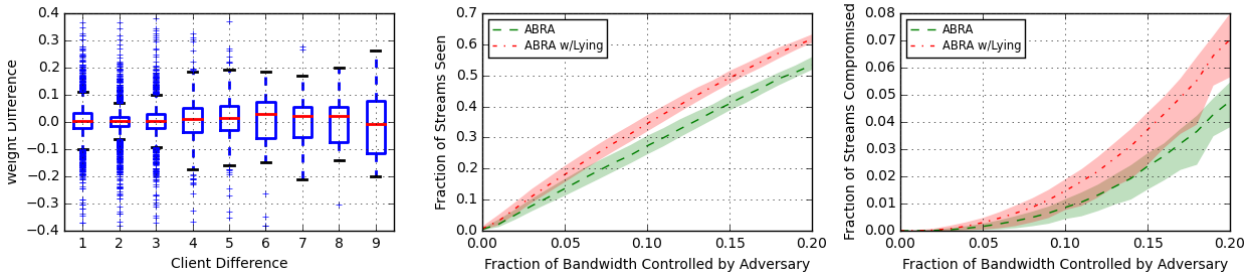
6.2 ABRA Performance

To evaluate the performance of Tor using ABRA for circuit selection, an experiment was configured to use ABRA with $g = 100ms$ and $w = 1s$. Additionally we ran experiments using congestion aware, latency aware, and latency weighted circuit selection. For congestion aware circuit selection three separate experiments were run, one using only passive probing, and two with clients actively probing circuits either every 5 or 60 seconds. Finally, an experiment was run with the central authority using the online algorithm to make circuit selection decisions for all clients. Figure 3 shows the CDF of the total relay bandwidth observed during the experiment. Congestion and latency aware network utilization, shown in Figures 3a and 3b, actually drop compared to vanilla Tor, pushing at best half as much data through the network. Clients in the congestion aware experiments were generally responding to out of date congestion information, with congested relays being over weighted by clients, causing those relays to become even more congested. Since latency aware circuit selection does not take into account relay bandwidth, low bandwidth relays with low latencies between each other are selected more often than they should be and become extremely congested.

While the congestion and latency aware algorithms performed worse than vanilla Tor, using both ABRA and the central authority for circuit selection produced increased network utilization. Figure 3c shows that ABRA and the central authority produced on average 14% and 20% better network utilization respectively when compared to vanilla Tor. Figure 4 looks at client performance using ABRA and centralized circuit selection. Download times universally improved, with some web clients performing downloads almost twice as fast, and bulk clients consistently seeing a 5-10% improvement. While all web clients had faster download times, about 20% saw even better results when using the central authority compared to ABRA for circuit selection. The only slight degradation in performance was with ABRA, where about 12-13% of downloads had a slower time to first byte compared to vanilla Tor. The central authority, however, consistently resulted in circuit selections that produced faster times to first byte across all client downloads.

6.3 Competitive Analysis

In this section we perform a competitive analysis, where the online algorithm run by the central authority is compared to the offline algorithm. We extracted all download start and end times from the vanilla experiment, along with the original circuits available to each download. Both the online algorithm (with the original and full circuit sets) and offline algorithm (seeded with original and pruned circuit sets) were run over the downloads to precompute a circuit selection. Experiments were then configured using each of



(a) (b) (c)

Figure 6: (a) weight difference on relays compared to the difference in number of bottleneck circuits on the relay (b) fraction of circuits seen by colluding relays (c) fraction of circuits where colluding relays appear as guard and exit

the computed circuit selections to analyze the exact network utilization impacts each of the circuit selections had.

Figures 5a shows total network bandwidth achieved with the online and offline algorithms when they use the full and pruned circuit sets respectively. Both algorithms produce almost identical network utilization, with median network bandwidth going from 148 MBps in vanilla Tor up to 259 MBps, a 75% increase. To see why we can look at relay capacity seen in Figure 5c. This looks at the percent of bandwidth being used on each relay compared to its configured `BandwidthRate`. Note that this can go higher than 100% because Tor also has a `BandwidthBurstRate` that allows a relay to temporarily send more than its `BandwidthRate` over short periods of time. This shows us that in the best algorithm runs relays are operating at very close to optimal utilization. Half of the time relays are using 90% or more of their configured bandwidth, compared to 20% of the time in vanilla Tor, meaning the centralized algorithms utilized resources that were otherwise being left idle.

Interestingly, while both algorithms had the highest performance gain when using larger circuit sets, they still produced improved performance when restricted to the original circuit sets as seen in Figure 5b. The online algorithm produced results close to those seen when using the full circuit set, with median network bandwidth at 248 MBps compared to 259 MBps for the full set. The offline algorithm, while still performing better than vanilla Tor, did not produce as much improvement when using the original circuit set, with network bandwidth at 206 MBps.

7. PRIVACY ANALYSIS

With the addition of gossip cells and using ABRA for circuit selection, there are some potential avenues for abuse an adversary could take advantage of to reduce client anonymity. In this section we cover each of these and examine how effective the methods are for an adversary.

7.1 Information Leakage

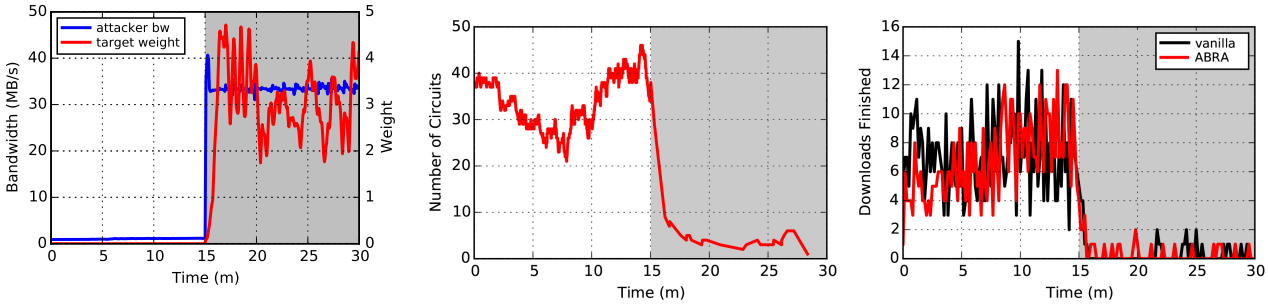
The first issue is that relays advertising their locally computed weight could leak information about other clients to an adversary. Mittal *et al.* [21] showed how an adversary could use throughput measurements to identify bottleneck relays in circuits. Relay weight values could be used similarly, where an adversary could attempt to correlate start and stop times of connections with the weight values of potential bottleneck relays. To examine how much information is leaked by the weight values, every time a relay sent a `GOSSIP` cell we recorded the weight of the relay and the number of active circuits using the relay. Then for every two consecutive `GOSSIP` cells sent by a relay we recorded the difference in weight and number of active circuits, to determine how the changes in these two values are correlated over a short period of time.

Figure 6a shows the distribution of weight differences across various changes in clients actively using the relay. Since we are interested in times when the relay is a bottleneck on new circuits, we excluded times when the weight difference was 0 as this is indicative that the relay was *not* a bottleneck on any of the new circuits. This shows an almost nonexistent correlation with an R^2 value of 0.00021. In the situation similar to the one outlined in [21] where an adversary is attempting to identify bottleneck relays used in a circuit, we are particularly interested in the situation where the number of active circuits using the relay as a bottleneck increases by 1. If there were large (maybe temporary) changes noticeable to an adversary they could identify the bottleneck relay. But as we can see in Figure 6a the distribution of weight changes when client difference is 1 is not significantly different from the distribution for larger client differences, meaning it would be extremely difficult to identify bottleneck relays by correlating weight changes.

7.2 Colluding Relays Lying

With relays self-reporting their locally calculated weights, adversarial relays could *lie* about their weight, consistently telling clients they have a weight of 0, increasing their chances of being on a selected circuit. Note that *construction* of circuits using ABRA is still unchanged, so the number of compromised circuits *built* by a client will not change; it is only when a client assigns streams to circuits that malicious relays could abuse `GOSSIP` cells to improve the probability of compromising a stream. Furthermore, this attack has a "self-damping" effect, in that attracting streams away from non-adversarial relays will decrease the bottleneck weight of those relays. Still, colluding relays, in an effort to increase the percent of circuits they are able to observe, could in a joint effort lie about their weights in an attempt to reduce anonymity of clients.

To determine how much of an effect such an attack would have, while running the experiment using ABRA for circuit selection we recorded the circuits available to the client, along with the respective weight and bandwidth information. A random set of relays were then marked as the "ad-



(a) **(b)** **(c)**

Figure 7: (a) attackers bandwidth and targets weight when an adversary is running the denial of service attack (b) number of clients using the target for a circuit (c) number of downloads completed by clients using the target relay before and after attack is started, shown both for vanilla Tor and using ABRA circuit selection

versary". For each circuit selection decision made, if the adversary had a relay on an available circuit we would rerun the circuit selection assuming all the adversarial relays had a reported weight of 0. We would then record the fraction of total bandwidth controlled by the adversary, the fraction of streams they saw with and without lying, and the fraction of *compromised* streams, with and without lying. A stream is considered compromised if the adversary controls both the guard and exit in its selected circuit. This process was repeated over 2,000,000 times to produce a range of expected streams seen and compromised as the fraction of bandwidth controlled by an adversary increases. Figures 6b and 6c shows the median fraction of streams observed bounded by the 10^{th} and 90^{th} percentile. As the adversary controls more bandwidth, and thus more relays that can lie about their weight, an adversary is able to observe about 10% more streams than they would have seen if they were not lying. Figure 6c shows the fraction of streams that are compromised with and without lying: when lying an adversary compromises roughly an additional 2-3% of streams. For example an adversary that controls 20% of the bandwidth compromises 6.9% with lying and 4.7% without. of streams compromised go from 4.7% to 6.9%. Note that this serves as an *upper* bound on the actual effect as this analysis is done statically. In actuality the non-adversarial relays would have a lower weight than we calculated as they would not be selected as often.

7.3 Denial of Service

While adversarial relays are limited in the number of extra streams they can observe by lying about their weight, they still could have the ability to reduce the chances that other relays are selected. To achieve this they would need to artificially inflate the weight of other relays in the network, preventing clients from selecting circuits that the target relays appear on. Recall that the local weight calculation is based on how many *bottleneck* circuits the relay estimates they are on. This means that an adversary cannot simply just create inactive circuits through the relay to inflate their weight, since those circuits would never be labeled as bottleneck. So to actually cause the weight to increase the adversary needs to actually send data through the circuits. To test the effectiveness of this attack, we configured an experiment to create 250 one-hop circuits through a target relay. After 15 minutes the one-hop circuits were activated, downloading as much data as they could. Note that we want as many circuits through the relay as possible to make their weight as large as possible. The relay weight is summed across all bottleneck circuits, $\sum bw(c_i)^{-1}$. If we have n one-hop circuits through a relay of bandwidth bw, each circuit will have a bandwidth of roughly $\frac{bw}{n}$, so the weight on the relay will be $\sum_1^n \left(\frac{bw}{n}\right)^{-1} = \frac{n^2}{bw}$.

Figure 7a looks at the weight of the target relay along with how much bandwidth the attacker is using, with the shaded region noting when the attack is active. We see that the attacker is able to push through close to the maximum bandwidth that the relay can handle, around 35 MB/s. When the circuits are active the weight spikes to almost 100 times what it was. After the attack is started the number plummets to almost 0, down from the 30-40 it was previously on. But while the attack does succeed in inflating the weight of the target, note that the adversary has to fully saturate the bandwidth of the target. Doing this in vanilla Tor will have almost the same effect, essentially running a denial of service by consuming all the available bandwidth. Figure 7c looks at the number of completed downloads using the target relay in both vanilla Tor and when using ABRA for circuit selection. Both experiments see the number of successful downloads drop to 0 while the attack is running. So even though the addition of the relay weight adds another mechanism that can be used to run a denial of service attack, the avenue (saturating bandwidth) is the same.

8. CONCLUSION

In this paper we introduce ABRA, the avoiding bottleneck relays algorithm. This algorithm lets relays estimate which circuits they are bottlenecks are on, allowing them to compute a weight that can be gossiped to clients, allowing for more coordinated circuit selection. ABRA is compared to congestion and latency aware circuit selection algorithms, showing that while these algorithms tend to actually *under* perform vanilla Tor, ABRA results in a 14% increase in network utilization. We show that the decentralized approach ABRA takes produces utilization close to what even a centralized authority can provide. Using competitive analysis we show that an online algorithm employed by a central authority matches a lower-bound offline genetic algorithm. Finally, we examine potential ways an adversary could abuse ABRA, finding that while information leakage is minimal,

there exist small increases in the percent of streams compromised based on the bandwidth controlled by an adversary acting maliciously.

Acknowledgments

This work was supported by the NSF under grant 1314637.

9. REFERENCES

[1] Shadow Homepage and Code Repositories. https://shadow.github.io/, https://github.com/shadow/.

[2] M. Akhoondi, C. Yu, and H. V. Madhyastha. LASTor: A Low-Latency AS-Aware Tor Client. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, May 2012.

[3] M. AlSabah, K. Bauer, and I. Goldberg. Enhancing tor's performance using real-time traffic classification. In *Proceedings of the 19th ACM conference on Computer and Communications Security (CCS 2012)*, October 2012.

[4] M. AlSabah, K. Bauer, I. Goldberg, D. Grunwald, D. McCoy, S. Savage, and G. Voelker. Defenestrator: Throwing out windows in tor. In *Proceedings of the 11th Privacy Enhancing Technologies Symposium (PETS 2011)*, July 2011.

[5] M. AlSabah and I. Goldberg. PCTCP: Per-Circuit TCP-over-IPsec Transport for Anonymous Communication Overlay Networks. In *Proceedings of the 20th ACM conference on Computer and Communications Security (CCS 2013)*, November 2013.

[6] M. Backes, A. Kate, S. Meiser, and E. Mohammadi. (nothing else) MATor(s): Monitoring the anonymity of tor's path selection. In *Proceedings of the 21th ACM conference on Computer and Communications Security (CCS 2014)*, November 2014.

[7] R. Dingledine, N. Mathewson, and P. Syverson. Tor: The second-generation onion router. In *Proceedings of the 13th USENIX Security Symposium*, August 2004.

[8] R. Dingledine and S. J. Murdoch. Performance improvements on tor or, why tor is slow and what we're going to do about it. *Online: http://www. torproject. org/press/presskit/2009-03-11-performance. pdf*, 2009.

[9] J. Geddes, R. Jansen, and N. Hopper. IMUX: Managing tor connections from two to infinity, and beyond. In *Proceedings of the 12th Workshop on Privacy in the Electronic Society (WPES)*, November 2014.

[10] D. Gopal and N. Heninger. Torchestra: Reducing interactive traffic delays over tor. In *Proceedings of the Workshop on Privacy in the Electronic Society (WPES 2012)*. ACM, October 2012.

[11] R. L. Graham, E. L. Lawler, J. K. Lenstra, and A. R. Kan. Optimization and approximation in deterministic sequencing and scheduling: a survey. *Annals of discrete mathematics*, 5:287–326, 1979.

[12] S. Herbert, S. J. Murdoch, and E. Punskaya. Optimising node selection probabilities in multi-hop m/d/1 queuing networks to reduce latency of tor. *Electronics Letters*, 50(17):1205–1207, 2014.

[13] R. Jansen, K. S. Bauer, N. Hopper, and R. Dingledine. Methodically modeling the tor network. In *CSET*, 2012.

[14] R. Jansen, J. Geddes, C. Wacek, M. Sherr, and P. Syverson. Never been kist: Tor's congestion management blossoms with kernel-informed socket transport. In *Proceedings of 23rd USENIX Security Symposium (USENIX Security 14)*, San Diego, CA, August 2014. USENIX Association.

[15] R. Jansen and N. Hopper. Shadow: Running Tor in a Box for Accurate and Efficient Experimentation. In *Proc. of the 19th Network and Distributed System Security Symposium*, 2012.

[16] R. Jansen, N. Hopper, and Y. Kim. Recruiting new Tor relays with BRAIDS. In A. D. Keromytis and V. Shmatikov, editors, *Proceedings of the 2010 ACM Conference on Computer and Communications Security (CCS 2010)*. ACM, October 2010.

[17] R. Jansen, A. Johnson, and P. Syverson. LIRA: Lightweight Incentivized Routing for Anonymity. In *Proceedings of the Network and Distributed System Security Symposium - NDSS'13*. Internet Society, February 2013.

[18] R. Jansen, P. Syverson, and N. Hopper. Throttling Tor Bandwidth Parasites. In *Proceedings of the 21st USENIX Security Symposium*, August 2012.

[19] G. F. Jenks. The data model concept in statistical mapping. *International yearbook of cartography*, 7(1):186–190, 1967.

[20] B. Jiang. Head/tail breaks: A new classification scheme for data with a heavy-tailed distribution. *The Professional Geographer*, 65(3):482–494, 2013.

[21] 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*, pages 215–226. ACM, 2011.

[22] T.-W. J. Ngan, R. Dingledine, and D. S. Wallach. Building Incentives into Tor. In R. Sion, editor, *Proceedings of Financial Cryptography (FC '10)*, January 2010.

[23] M. F. Nowlan, N. Tiwari, J. Iyengar, S. O. Aminy, and B. Ford. Fitting square pegs through round pipes: Unordered delivery wire-compatible with tcp and tls. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 28–28. USENIX Association, 2012.

[24] M. F. Nowlan, D. Wolinsky, and B. Ford. Reducing latency in tor circuits with unordered delivery. In *USENIX Workshop on Free and Open Communications on the Internet (FOCI)*, 2013.

[25] E. Parzen. On estimation of a probability density function and mode. *The annals of mathematical statistics*, 33(3):1065–1076, 1962.

[26] J. Reardon. Improving Tor using a TCP-over-DTLS tunnel. Master's thesis, University of Waterloo, September 2008.

[27] M. Rosenblatt et al. Remarks on some nonparametric estimates of a density function. *The Annals of Mathematical Statistics*, 27(3):832–837, 1956.

[28] T. Roughgarden. *Selfish routing*. PhD thesis, Cornell University, 2002.

[29] T. Roughgarden. *Selfish routing and the price of anarchy*, volume 174. MIT press Cambridge, 2005.

[30] M. Sherr, M. Blaze, and B. T. Loo. Scalable link-based relay selection for anonymous routing. In *Privacy Enhancing Technologies*, pages 73–93. Springer, 2009.

[31] M. Sherr, A. Mao, W. R. Marczak, W. Zhou, B. T. Loo, and M. A. Blaze. A^3: An extensible platform for application-aware anonymity. 2010.

[32] R. Snader and N. Borisov. Eigenspeed: secure peer-to-peer bandwidth evaluation. In *IPTPS*, page 9, 2009.

[33] R. Snader and N. Borisov. Improving security and performance in the tor network through tunable path selection. *Dependable and Secure Computing, IEEE Transactions on*, 8(5):728–741, 2011.

[34] C. Tang and I. Goldberg. An improved algorithm for Tor circuit scheduling. In A. D. Keromytis and

V. Shmatikov, editors, *Proceedings of the 2010 ACM Conference on Computer and Communications Security (CCS 2010)*. ACM, October 2010.

[35] C. Wacek, H. Tan, K. Bauer, and M. Sherr. An Empirical Evaluation of Relay Selection in Tor. In *Proceedings of the Network and Distributed System Security Symposium - NDSS'13*. Internet Society, February 2013.

[36] T. Wang, K. Bauer, C. Forero, and I. Goldberg. Congestion-aware Path Selection for Tor. In *Proceedings of Financial Cryptography and Data Security (FC'12)*, February 2012.

[37] Y. Yang, J. K. Muppala, and S. T. Chanson. Quality of service routing algorithms for bandwidth-delay constrained applications. In *Network Protocols, 2001. Ninth International Conference on*, pages 62–70. IEEE, 2001.

APPENDIX

A. DELAY-WEIGHTED CAPACITY ROUTING

In the algorithm, the network is represented as an undirected graph, where each vertex is a router and edges represent links between routers, with the bandwidth and delay of each link assigned to the edge. For an ingress-egress pair of routers (s,t), the algorithm continually extracts a least delay path LP_i, adds it to the set of all least delay paths LP, and then removes all edges in LP_i from the graph. This step is repeated until no paths exist between s and t in the graph, leaving us with a set of least delay paths $LP = \{LP_1, LP_2, \ldots, LP_k\}$.

For each path LP_i we have the residual bandwidth B_i which is the minimum bandwidth across all links, and the end-to-end delay D_i across the entire path. The delay-weighted capacity (DWC) of the ingress-egress pair (s,t) is then defined as $DWC = \sum_{i=1}^{k} \frac{B_i}{D_i}$. The link that determines the bandwidth for a path is the *critical link*, with the set of all critical links represented by $C = \{C_1, C_2, \ldots, C_k\}$. The basic idea is that when picking a path, we want to avoid critical links as much as possible. To do so, each link is assigned a weight based on how many times it is a critical link in a path: $w_l = \sum_{l \in C_i} \alpha_i$. The alpha value can take on one of three functions: (1) Number of times a link is critical ($\alpha_i = 1$) (2) The overall delay of the path ($\alpha_i = \frac{1}{D_i}$) (3) The delay and bandwidth of the path ($\alpha_i = \frac{1}{B_i \cdot D_i}$) . With each link assigned a weight, the routing algorithm simply chooses the path with the lowest sum of weights across all links in the path. Once a path is chosen, the bandwidth of the path is subtracted from the available bandwidth of all links in the path. Any links with no remaining available bandwidth are removed from the graph and the next path can be selected.

B. CLUSTERING ALGORITHMS

Jenks Natural Breaks: The Jenks natural breaks [19] clustering algorithm attempts to cluster one-dimensional data into classes that minimizes the in-class variance. The function takes in a one-dimensional array of data and the n classes that the data should be clustered into. It returns the breaks $[b_1, b_2), [b_2, b_3), ..., [b_n, b_{n+1}]$ and a goodness of variance fit (GVF) $var \in [0,1]$, where higher variances indicate better fits. For clustering circuits we can use the Jenks algorithm in two different ways. First is to cluster the data into 2 classes, with circuits in the $[b_1, b_2)$ range classified as non-bottlenecks and those in $[b_2, b_3]$ classified as bottlenecks. Second is we can keep incrementing the number of classes we cluster the circuits into until the GVF value passes some threshold τ. Once the threshold is passed we then classify all circuits in the $[b_n, b_{n+1}]$ range as bottlenecks and everyone else as non-bottlenecks. The circuits in the final head class are then labeled bottleneck circuits. Finally we use a kernel density estimator [25, 27] to fit a multimodal distribution to every circuits bandwidth history. With the distribution we compute the set of local minima and classify every circuit with a bandwidth higher than the last minima as a bottleneck.

Algorithm 1 Head/Tail clustering algorithm

```
function HeadTail(data, threshold)
    m ← sum(data)/len(data)
    head ← {d ∈ data | d ≥ m}
    if len(head)/len(data) < threshold then
        head ← HeadTail(head, threshold)
    end if
    return head
end function
```

Head/Tail: The head/tail clustering algorithm [20] is useful when the underlying data has a long tail, which could be useful for bottleneck identification, as we expect to have a tight clustering around the bottlenecks with other circuits randomly distributed amongst the lower bandwidth values. The algorithm first splits the data into two sets, the tail set containing all values less than the arithmetic mean, and everything greater to or equal to the mean is put in the head set. If the percent of values that ended up in the head set is less than some threshold, the process is repeated using the head set as the new data set. Once the threshold is passed the function returns the very last head set as the head cluster of the data. This algorithm is shown in Algorithm 1. For bottleneck identification we simply pass in the circuit bandwidth data and the head cluster returned contains all the bottleneck circuits.

Kernel Density Estimator: The idea behind the kernel density estimator [25, 27] is we are going to try and fit a multimodal distribution based on a Gaussian kernel to the circuits. Instead of using the bandwidth estimate for each circuit, the estimator takes as input the entire bandwidth history seen across all circuits, giving the estimator more data points to build a more accurate density estimate. For the kernel bandwidth we initially use the square root of the mean of all values. Once we have a density we compute the set of local minima $\{m_1, ..., m_n\}$ and classify every circuit with bandwidth above m_n as a bottleneck. If the resulting density estimate is unimodal and we do not have *any* local minima, we repeat this process, halving the kernel bandwidth until we get a multimodal distribution.

C. BANDWIDTH ALGORITHM PROOF

Let R be the relay selected with B bandwidth and C circuits. Let R' be a different relay with B' bandwidth and C' circuits. By definition R is selected such that $\frac{B}{C} \leq \frac{B'}{C'}$. When iterating through the C circuits let n be the number that R' is on. Note that this means that $n \leq C'$. After the circuits have been iterated through and operations per-

formed, R' will have $B' - \frac{B}{C} \cdot n$ bandwidth left with $C' - n$ circuits.

Assume that R' has 0 bandwidth afterwards, so $B' - \frac{B}{C} \cdot n = 0$. We want to show this means that R' is on no more circuits so that $C' - n = 0$. We have

$$B' - \frac{B}{C} \cdot n = 0 \Rightarrow B' = \frac{B}{C} \cdot n \Rightarrow n = \frac{B' \cdot C}{B}$$

So that means that the number of circuits R' is left on is

$$\begin{aligned} C' - n &= C' - C - \frac{B' \cdot C}{B} = \frac{B \cdot C'}{B} - \frac{B' \cdot C}{B} \\ &= \frac{B \cdot C' - B' \cdot C}{B} \end{aligned}$$

However, R was picked such that

$$\frac{B}{C} \leq \frac{B'}{C'} \Rightarrow B \cdot C' \leq B' \cdot C \Rightarrow B \cdot C' - B' \cdot C \leq 0$$

This gives us

$$C' - n = \frac{B \cdot C' - B' \cdot C}{B} \leq 0$$

since we know $B > 0$ and

$$n \leq C' \Rightarrow 0 \leq C' - n$$

which implies that $0 \leq C' - n \leq 0 \Rightarrow C' - n = 0$.

D. ALGORITHMS

Algorithm 2 Calculate relay weight and active circuit bandwidth

```
1: function CalcCircuitBW(activeCircuits)
2:     activeRelays ← GetRelays(activeCircuits)
3:     while not activeCircuits.empty() do
4:         r ← GetBottleneckRelay(activeRelays)
5:         r.weight ← 1/r.bw
6:         circuits ← GetCircuits(activeCircuits, r)
7:         circuitBW ← r.bw/circuits.len()
8:         for c ∈ circuits do
9:             c.bw ← circuitBW
10:            for circRelay ∈ c.relays do
11:                circRelay.bw −= c.bw
12:                if circRelay.bw = 0 then
13:                    activeRelays.remove(circRelay)
14:                end if
15:            end for
16:            activeCircuits.remove(c)
17:        end for
18:    end while
19: end function
```

Algorithm 3 Compute total bandwidth for downloads

```
1: function CalcTotalBW(downloads)
2:     start, end ← GetTimeInterval(downloads)
3:     bandwidth ← 0
4:     time ← start
5:     while time ≤ end do
6:         circuits ← GetActiveCircs(downloads, time)
7:         CalcCircuitBW(circuits)
8:         for circuit ∈ circuits do
9:             bandwidth += circuit.bw
10:        end for
11:        time ← time + tick
12:    end while
13:    return bandwidth
14: end function
```

Algorithm 4 Generate pruned circuit set

```
1: function BuildPrunedSet(relays)
2:     circuits ← List()
3:     while TRUE do
4:         if relays.len() > 3 then
5:             break
6:         end if
7:         if relays.numExits() > 0 then
8:             break
9:         end if
10:        relays.sortByBW()
11:        exit ← relays.getFirstExit()
12:        middle ← relays.getFirstNonExit()
13:        guard ← relays.getFirstNonExit()
14:        if middle == Null then
15:            middle ← relays.getFirstExit()
16:        end if
17:        if guard == Null then
18:            guard ← relays.getFirstExit()
19:        end if
20:        circuits.append(guard, middle, exit)
21:        bw ← min(guard.bw, middle.bw, exitbw)
22:        guard.bw ← guard.bw − bw
23:        middle.bw ← middle.bw − bw
24:        exit.bw ← exit.bw − bw
25:        if guard.bw == 0 then
26:            relays.remove(guard)
27:        end if
28:        if middle.bw == 0 then
29:            relays.remove(middle)
30:        end if
31:        if exit.bw == 0 then
32:            relays.remove(exit)
33:        end if
34:    end while
35:    return circuits
36: end function
```