Throttling Tor Bandwidth Parasites

Rob Jansen Paul Syverson Nicholas Hopper
U.S. Naval Research Laboratory *University of Minnesota*
{*rob.g.jansen, paul.syverson*}*@nrl.navy.mil* *hopper@cs.umn.edu*

Abstract

Tor is vulnerable to network congestion and performance problems due to bulk data transfers. A large fraction of the available network capacity is consumed by a small percentage of Tor users, resulting in severe service degradation for the majority. Bulk users continuously drain relays of excess bandwidth, creating new network bottlenecks and exacerbating the effects of existing ones. While this problem may currently be attributed to rational users utilizing the network, it may also be exploited by a relatively low-resource adversary using similar techniques to contribute to a network denial of service (DoS) attack. Degraded service discourages the use of Tor, affecting both Tor's client diversity and anonymity.

Equipped with mechanisms from communication networks, we design and implement three Tor-specific algorithms that throttle bulk transfers to reduce network congestion and increase network responsiveness. Unlike existing techniques, our algorithms adapt to network dynamics using only information local to a relay. We experiment with full-network deployments of our algorithms under a range of light to heavy network loads. We find that throttling results in significant improvements to web client performance while mitigating the negative effects of bulk transfers. We also analyze how throttling affects anonymity and compare the security of our algorithms under adversarial attack. We find that throttling reduces information leakage compared to unthrottled Tor while improving anonymity against realistic adversaries.

1 Introduction

The Tor [19] anonymity network was developed in an attempt to improve anonymity on the Internet. Onion Routing [23, 48] serves as the cornerstone for Tor's overlay network design. Tor *clients* encrypt messages in several "layers" while packaging them into 512-byte packets called *cells*, and send them through a collection of *relays* called a *circuit*. Each relay decrypts its layer and forwards the message to the next relay in the circuit. The last relay forwards the message to the user-specified destination. Each relay can determine only its predecessor and successor in the path from source to destination, preventing any single relay from linking the sender and receiver. Clients choose their first relay from a small set of *entry guards* [44, 59] in order to help defend against passive logging attacks [58]. Traffic analysis is still possible [8, 22, 28, 30, 39, 42, 46, 49], but slightly complicated by the fact that each relay simultaneously services multiple circuits.

Tor relays are run by volunteers located throughout the world and service hundreds of thousands of Tor clients [37] with high bandwidth demands. A relay's utility to Tor is dependent on both the bandwidth *capacity* of its host network and the bandwidth *restrictions* imposed by its operator. Although bandwidth donations vary widely, the majority of relays offer less than 100 KiB/s and may become bottlenecks when chosen for a circuit. Bandwidth bottlenecks lead to network congestion and impair client performance.

Bottlenecks are further aggravated by bulk users, which make up roughly five percent of connections and forty percent of the bytes transferred through the network [38]. Bulk traffic increases network-wide congestion and punishes interactive users as they attempt to browse the web and run `SSH` sessions. Bulk traffic also constitutes a simple denial of service (DoS) attack on the network as a whole: with nothing but a moderate number of bulk clients, an adversary can intentionally significantly degrade the performance of the entire Tor network for most users. This is a malicious attack as opposed to an opportunistic use of resources without regard for the impact on legitimate users, and could be used by censors [16] to discourage use of Tor. Bulk traffic effectively averts potential users from Tor, decreasing both Tor's client diversity and anonymity [10, 18].

There are three general approaches to alleviate Tor's performance problems: increase network capacity; optimize resource utilization; and reduce network load.

Increasing Capacity. One approach to reducing bottlenecks and improving performance is to add additional bandwidth to the network from new relays. Previous work has explored recruiting new relays by offering performance incentives to those who contribute [32, 41, 43]. While these approaches show potential, they have not been deployed due to a lack of understanding of the anonymity and economic implications they would impose on Tor and its users. It is unclear how an incentive

scheme will affect users' anonymity and motivation to contribute: Acquisti *et al.* [6] discuss how differentiating users by performance may reduce anonymity while competition may reduce the sense of community and convince users that contributions are no longer warranted.

New high-bandwidth relays may also be added by the Tor Project [4] or other organizations. While effective at improving network capacity, this approach is a short-term solution that does not scale. As Tor increases speed and bandwidth, it will likely attract more users. More significantly, it will attract more high-bandwidth and BitTorrent users, resulting in a *Tragedy of the Commons* [26] scenario: the bulk users attracted to the faster network will continue to leech the additional bandwidth.

Optimizing Resource Utilization. Another approach to improving performance is to better utilize the available network resources. Tor's path selection algorithm ignores the slowest small fraction of relays while selecting from the remaining relays in proportion to their available bandwidth. The path selection algorithm also ignores circuits with long build times [12], removing the worst of bottlenecks and improving usability. Congestion-aware path selection [57] is another approach that aims to balance load by using opportunistic and active client measurements while building paths. However, low bandwidth relays must still be chosen for circuits to mitigate anonymity problems, meaning there are still a large number of circuits with tight bandwidth bottlenecks.

Tang and Goldberg previously explored modifications to the Tor circuit scheduler in order to prioritize bursty (i.e. web) traffic over bulk traffic using an exponentially-weighted moving average (EWMA) of relayed cells [52]. Early experiments show small improvements at a single relay, but full-network experiments indicate that the new scheduler has an insignificant effect on performance [31]. It is unclear how performance is affected when deployed to the live Tor network. This scheduling approach attempts to shift network load to better utilize the available bandwidth, but does not reduce bottlenecks introduced by the massive amount of bulk traffic currently plaguing Tor.

Reducing Load. All of the previously discussed approaches attempt to increase performance, but none of them directly address or provide adequate defense against performance degradation problems created by bulk traffic clients. In this paper, we address these by adaptively throttling bulk data transfers at the client's entry into the Tor network.

We emphasize that throttling is *fundamentally different* than scheduling, and the distinction is important in the context of the Tor network. Schedulers optimize the utilization of available bandwidth by following policies set by the network engineer, allowing the enforcement of fairness among flows (e.g. max-min fairness [24, 34] or proportional fairness [35]). However, throttling may *under-utilize* local bandwidth resources by intentionally imposing restrictions on clients' throughput to reduce aggregate network load.

By reducing bulk client throughput in Tor, we effectively reduce the bulk data transfer rate through the network, resulting in *fewer bottlenecks* and a less congested, more responsive Tor network that can better handle the burstiness of web traffic. Tor has recently implemented token buckets, a classic traffic shaping mechanism [55], to statically (non-adaptively) throttle client-to-guard connections at a given rate [17], but currently deployed configurations of Tor do not enable throttling by default. Unfortunately, the throttling algorithm implemented in Tor requires static configuration of throttling parameters: the Tor network must determine network-wide settings that work well and update them as the network changes. Further, it is not possible to automatically tune each relay's throttling configuration with the current algorithm.

Contributions. To the best of our knowledge, we are the first to explore throttling algorithms that *adaptively adjust* to the fluctuations and dynamics of Tor and each relay independently without the need to adjust parameters as the network changes. We also perform the first detailed investigation of the performance and anonymity implications of throttling Tor client connections.

In Section 3, we introduce and test three algorithms that dynamically and adaptively throttle Tor clients using a basic token bucket rate-limiter as the underlying throttling mechanism. Our new adaptive algorithms use local relay information to dynamically select which connections get throttled and to adjust the rate at which those connections are throttled. Adaptively tuned throttling mechanisms are paramount to our algorithm designs in order to avoid the need to re-evaluate parameter choices as network capacity or relay load changes. Our *bit-splitting* algorithm throttles each connection at an adaptively adjusted, but reserved and equal portion of a guard node's bandwidth, our *flagging* algorithm aggressively throttles connections that have historically exceeded the statistically fair throughput, and our *threshold* algorithm throttles connections above a throughput quantile at a rate represented by that quantile.

We implement our algorithms in Tor[1] and test their effectiveness at improving performance in large scale, full-network deployments. Section 4 compares our algorithms to static (non-adaptive) throttling under a varied range of network loads. We find that the effectiveness of static throttling is highly dependent on network load and configuration whereas our adaptive algorithms work well under various loads with no configuration changes or parameter maintenance: web client performance was

[1] Software patches for our algorithms have been made publicly available to the community [5].

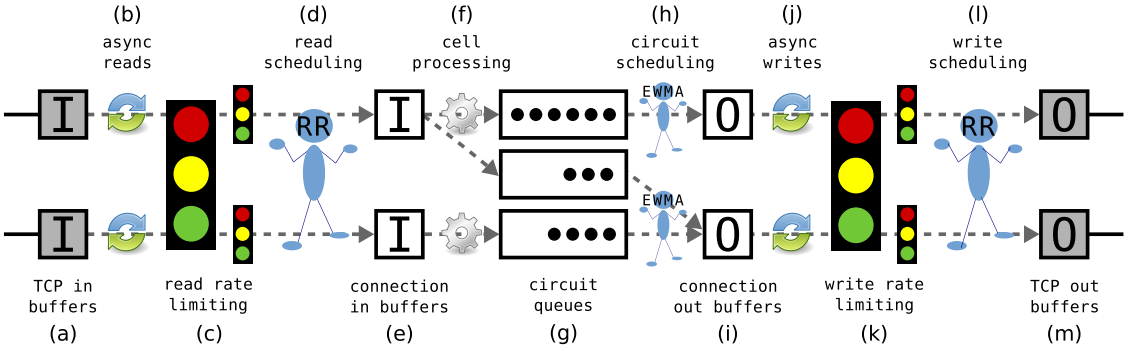


Figure 1: A Tor relay's internal architecture.

improved for every parameter setting we tested. We conclude that throttling is an effective approach to achieve a more responsive network.

Having shown that our adaptive throttling algorithms provide significant performance benefits for web clients and have a profound impact on network responsiveness, Section 5 analyzes the security of our algorithms under adversarial attack. We discuss several realistic attacks on anonymity and compare the information leaked by each algorithm relative to unthrottled Tor. Against intuition, we find that throttling clients reduces information leakage and improves network anonymity while minimizing the false positive impact on honest users.

2 Background

This section discusses Tor's internal architecture, shown in Figure 1, to facilitate an understanding of how internal processes affect client traffic flowing through a Tor relay.

Multiplexed Connections. All relays in Tor communicate using pairwise TCP *connections*, i.e. each relay forms a single TCP connection to each other relay with which it communicates. Since a pair of relays may be communicating data for several *circuits* at once, all circuits between the pair are multiplexed over their single TCP connection. Each circuit may carry traffic for multiple services or *streams* that a user may be accessing. TCP offers reliability, in-order delivery of packets between relays, and potentially unfair kernel-level congestion control when multiplexing connections [47]. The distinction between and interaction of connections, circuits, and streams is important for understanding Tor.

Connection Input. Tor uses libevent [1] to handle input and output to and from kernel TCP buffers. Tor registers sockets that it wants to read with libevent and configures a notification callback function. When data arrives at the kernel TCP input buffer (Figure 1a), libevent learns about the active socket through its polling interface and asynchronously executes the corresponding callback (Figure 1b). Upon execution, the read callback determines read eligibility using token buckets.

Token buckets are used to rate-limit connections. Tor fills the buckets as defined by configured bandwidth limits in one-second intervals while tokens are removed from the buckets as data is read, although changing that interval to improve performance is currently being explored [53]. There is a global read bucket that limits bandwidth for reading from all connections as well as a separate bucket for throttling on a per-connection basis (Figure 1c). A connection may ignore a read event if either the global bucket or its connection bucket is empty. In practice, the per-connection token buckets are only utilized for edge (non-relay) connections. Per-connection throttling reduces network congestion by penalizing noisy connections, such as bulk transfers, and generally leads to better performance [17].

When a TCP input buffer is eligible for reading, a round-robin (RR) scheduling mechanism is used to read the smaller of 16 KiB and $\frac{1}{8}$ of the global token bucket size per connection (Figure 1d). This limit is imposed in an attempt at fairness so that a single connection can not consume all the global tokens on a single read. However, recent research shows that input/output scheduling leads to unfair resource allocations [54]. The data read from the TCP buffer is placed in a per-connection application input buffer for processing (Figure 1e).

Flow Control. Tor uses an end-to-end flow control algorithm to assist in keeping a steady flow of cells through a circuit. Clients and exit relays constitute the *edges* of a circuit: each are both an ingress and egress point for data traversing the Tor network. Edges track data flow for both circuits and streams using cell counters called *windows*. An ingress edge decrements the corresponding stream and circuit windows when sending cells, stops reading from a stream when its stream window reaches zero, and stops reading from all streams multiplexed over a circuit when the circuit window reaches zero. Windows are incremented and reading resumes upon receipt of `SENDME` acknowledgment cells from egress edges.

By default, circuit windows are initialized to 1000 cells (500 KiB) and stream windows to 500 cells (250 KiB). Circuit SENDMEs are sent to the ingress edge after the egress edge receives 100 cells (50 KiB), allowing the ingress edge to read, package, and forward 100 additional cells. Stream SENDMEs are sent after receiving 50 cells (25 KiB) and allow an additional 50 cells. Window sizes can have a significant effect on performance and recent work suggests an algorithm for dynamically computing them [7].

Cell Processing and Queuing. Data is immediately processed as it arrives in connection input buffers (Figure 1f) and each cell is either encrypted or decrypted depending on its direction through the circuit. The cell is then switched onto the circuit corresponding to the next hop and placed into the circuit's first-in-first-out (FIFO) queue (Figure 1g). Cells wait in circuit queues until the circuit scheduler selects them for writing.

Scheduling. When there is space available in a connection's output buffer, a relay decides which of several multiplexed circuits to choose for writing. Although historically this was done using round-robin, a new exponentially-weighted moving average (EWMA) scheduler was recently introduced into Tor [52] and is currently used by default (Figure 1h). EWMA records the number of packets it schedules for each circuit, exponentially decaying packet counts over time. The scheduler writes one cell at a time chosen from the circuit with the lowest packet count and then updates the count. The decay means packets sent more recently have a higher influence on the count while bursty traffic does not significantly affect scheduling priorities.

Connection Output. A cell that has been chosen and written to a connection output buffer (Figure 1i) causes an activation of the write event registered with libevent for that connection. Once libevent determines the TCP socket can be written, the write callback is asynchronously executed (Figure 1j). Similar to connection input, the relay checks both the global write bucket and per-connection write bucket for tokens. If the buckets are not empty, the connection is eligible for writing (Figure 1k) and again is allowed to write the smaller of 16 KiB and $\frac{1}{8}$ of the global token bucket size per connection (Figure 1l). The data is written to the kernel-level TCP buffer (Figure 1m) and sent to the next hop.

3 Throttling Client Connections

Client performance in Tor depends heavily on the traffic patterns of others in the system. A small number of clients performing bulk transfers in Tor are the source of a large fraction of total network traffic [38]. The overwhelming load these clients place on the network increases congestion and creates additional bottlenecks, causing interactive applications, such as instant messaging and remote SSH sessions, to lose responsiveness.

Figure 2: Throttling occurs at the connection between the client and guard to capture all streams to various destinations.

This section explores client throttling as a mechanism to prevent bulk clients from overwhelming the network. Although a relay may have enough bandwidth to handle all traffic locally, bulk clients that continue producing additional traffic cause bottlenecks at other low-capacity relays. The faster a bulk downloader gets its data, the faster it will pull more into the network. Throttling bulk and other high-traffic clients prevents them from pushing or pulling too much data into the network too fast, reducing these bottlenecks and improving performance for the majority of users. Therefore, interactive applications and Tor in general will become much more usable, attracting new users who improve client diversity and anonymity.

We emphasize that throttling algorithms are not a replacement for congestion control or scheduling algorithms, although each approach may cooperate to achieve a common goal. Scheduling algorithms are used to *manage the utilization of bandwidth*, throttling algorithms *reduce the aggregate network load*, and congestion control algorithms attempt to do both. The distinction between congestion control and throttling algorithms is subtle but important: congestion control reduces *circuit load* while attempting to maximize network utilization, whereas throttling reduces *network load* in an attempt to improve circuit performance by explicitly under-utilizing connections to bulk clients using too many resources. Each approach may independently affect performance, and they may be combined to improve the network.

3.1 Static Throttling

Recently, Tor introduced the functionality to allow entry guards to throttle connections to clients [17] (see Figure 2). This client-to-guard connection is targeted because all client traffic (using this guard) will flow over this connection regardless of the number of streams or the destination associated with each.[2] The implementation uses a token bucket for each connection in addition to the global token bucket that already limits the total amount of bandwidth used by a relay. The size of the per-connection token buckets can be specified using the PerConnBWBurst configuration option, and the bucket refill rate can be specified by configuring the PerConnBWRate. The configured throttling rate en-

[2]This work does not consider modified Tor clients.

sures that all client-to-guard connections are throttled to the specified long-term-average throughput while the configured burst allows deviations from the throttling rate to account for bursty traffic. The configuration options provide a static throttling mechanism: Tor will throttle all connections using these values until directed otherwise. Note that Tor does not enable or configure static throttling by default.

While static throttling is simple, it has two main drawbacks. First, static throttling requires constant monitoring and measurements of the Tor network to determine which configurations work well and which do not in order to be effective. We have found that there are many configurations of the algorithm that cause no change in performance, and worse, there are configurations that harm performance for interactive applications [33]. This is the opposite of what throttling is attempting to achieve. Second, it is not possible under the current algorithm to auto-tune the throttling parameters for each Tor relay. Configurations that appear to work well for the network as a whole might not necessarily be tuned for a given relay (we will show that this is indeed the case in Section 4). Each relay has very different capabilities and load patterns, and therefore may require different throttling configurations to be most useful.

3.2 Adaptive Throttling

Given the drawbacks of static throttling, we now explore and present three new algorithms that adaptively adjust throttling parameters according to local relay information. This section details our algorithms while Section 4 explores their effect on client performance and Section 5 analyzes throttling implications for anonymity.

There are two main issues to consider when designing a client throttling algorithm: *which connections* to throttle and at *what rate* to throttle them. The approach discussed above in Section 3.1 throttles *all* client connections at the *statically* specified rate. Each of our three algorithms below answers these questions *adaptively* by considering information local to each relay. Note that our algorithms dynamically adjust the `PerConnBWRate` while keeping a constant `PerConnBWBurst`.[3]

Bit-splitting. A simple approach to adaptive throttling is to split a guard's bandwidth equally among all active client connections and throttle them all at this *fair split rate*. The `PerConnBWRate` will therefore be adjusted as new connections are created or old connections are destroyed: more connections will result in lower rates. No connection will be able to use more than its allotted share of bandwidth unless it has unused tokens in its bucket. Inspired by Quality of Service (QoS) work from communication networks [11, 50, 60], bit-splitting will prevent bulk clients from unfairly consuming bandwidth and ensure that there is a minimum "reserved" bandwidth for clients of all types.

Algorithm 1 Throttling clients by splitting bits.

```
1:  B ← getRelayBandwidth()
2:  L ← getConnectionList()
3:  N ← L.length()
4:  if N > 0 then
5:      splitRate ← B/N
6:      for i ← 1 to N do
7:          if L[i].isClientConnection() then
8:              L[i].throttleRate ← splitRate
9:          end if
10:     end for
11: end if
```

Note that Internet Service Providers employ similar techniques to throttle their customers, however, their client base is much less dynamic than the connections an entry guard handles. Therefore, our adaptive approach is more suitable to Tor. We include this algorithm in our analysis of throttling to determine what is possible with such a simple approach.

Flagging Unfair Clients. The bit-splitting algorithm focuses on adjusting the throttle rate and applying this to all client connections. Our next algorithm takes the opposite approach: configure a static throttling rate and adjust which connections get throttled. The intuition behind this approach is that if we can properly identify the connections that use too much bandwidth, we can throttle them in order to maximize the benefit we gain per throttled connection. Therefore, our flagging algorithm attempts to classify and throttle bulk traffic while it avoids throttling web clients.

Since deep packet inspection is not desirable for privacy reasons, and is not possible on encrypted Tor traffic, we instead draw upon existing *statistical fingerprinting* classification techniques [14, 29, 36] that classify traffic solely on its statistical properties. When designing the flagging algorithm, we recognize that Tor already contains a statistical throughput measure for scheduling traffic on *circuits* using an exponentially-weighted moving average (EWMA) of recently sent cells [52]. We can use the same statistical measure on client *connections* to classify and throttle bulk traffic.

The flagging algorithm, shown in Algorithm 2, requires that each guard keeps an EWMA of the number of recently sent cells per client connection. The per-connection cell EWMA is computed in much the same way as the per-circuit cell EWMA: whenever the cir-

[3] Our experiments [33] indicate that a 2 MiB burst is ideal as it allows directory requests to be downloaded unthrottled during bootstrapping while also throttling bulk traffic relatively quickly. The burst may need to be increased if the directory information grows beyond 2 MiB.

Algorithm 2 Throttling clients by flagging bulk connections, considering a moving average of throughput.

Require: $flagRate, \mathcal{P}, \mathcal{H}$
1: $B \leftarrow getRelayBandwidth()$
2: $L \leftarrow getConnectionList()$
3: $N \leftarrow L.length()$
4: $\mathcal{M} \leftarrow getMetaEWMA()$
5: **if** $N > 0$ **then**
6: $splitRate \leftarrow \frac{B}{N}$
7: $\mathcal{M} \leftarrow \mathcal{M}.increment(\mathcal{H}, splitRate)$
8: **for** $i \leftarrow 1$ to N **do**
9: **if** $L[i].isClientConnection()$ **then**
10: **if** $L[i].EWMA > \mathcal{M}$ **then**
11: $L[i].flag \leftarrow True$
12: $L[i].throttleRate \leftarrow flagRate$
13: **else if** $L[i].flag = True \wedge$ $L[i].EWMA < \mathcal{P} \cdot \mathcal{M}$ **then**
14: $L[i].flag \leftarrow False$
15: $L[i].throttleRate \leftarrow infinity$
16: **end if**
17: **end if**
18: **end for**
19: **end if**

Algorithm 3 Throttling clients considering the loudest threshold of connections.

Require: $\mathcal{T}, \mathcal{R}, \mathcal{F}$
1: $L \leftarrow getClientConnectionList()$
2: $N \leftarrow L.length()$
3: **if** $N > 0$ **then**
4: $selectIndex \leftarrow floor(\mathcal{T} \cdot N)$
5: $L \leftarrow reverseSortEWMA(L)$
6: $thresholdRate \leftarrow L[selectIndex].$ $getMeanThroughput(\mathcal{R})$
7: **if** $thresholdRate < \mathcal{F}$ **then**
8: $thresholdRate \leftarrow \mathcal{F}$
9: **end if**
10: **for** $i \leftarrow 1$ to N **do**
11: **if** $i \leq selectIndex$ **then**
12: $L[i].throttleRate \leftarrow thresholdRate$
13: **else**
14: $L[i].throttleRate \leftarrow infinity$
15: **end if**
16: **end for**
17: **end if**

cuit's cell counter is incremented, so is the cell counter of the connection to which that circuit belongs. Note that clients can not affect others' per-connection EWMA since all of a client's circuits are multiplexed over a single throttled guard-to-client connection.[4] The per-connection EWMA is enabled and configured independently of its circuit counterpart.

We rely on the observation that bulk connections will have higher EWMA values than web connections since bulk clients are steadily transferring data while web clients "think" between each page download. Using this to our advantage, we can flag connections as containing bulk traffic as follows. Each relay keeps a single separate meta-EWMA $\mathcal{M}$ of cells transferred. $\mathcal{M}$ is adjusted by calculating the fair bandwidth split rate as in the bit-splitting algorithm, and tracking its EWMA over time. $\mathcal{M}$ does not correspond with any real traffic, but represents the upper bound of a connection-level EWMA if a connection were continuously sending only its fair share of traffic through the relay. Any connection whose EWMA exceeds $\mathcal{M}$ is flagged as containing bulk traffic and penalized by being throttled.

There are three main parameters for the algorithm. As mentioned above, a per-connection half-life $\mathcal{H}$ allows configuration of the connection-level half-life independent of that used for circuit scheduling. $\mathcal{H}$ affects how long the algorithm remembers the amount of data a connection has transferred, and has precisely the same meaning as the circuit priority half-life [52]. Larger half-life values increase the ability to differentiate bulk from web connections while smaller half-life values make the algorithm more immediately reactive to throttling bulk connections. We would like to allow for a specification of the length of each penalty once a connection is flagged in order to recover and stop throttling connections that may have been incorrectly flagged. Therefore, we introduce a penalty fraction parameter $\mathcal{P}$ that affects how long each connection remains in a flagged and throttled state. If a connection's cell count EWMA falls below $\mathcal{P} \cdot \mathcal{M}$, its flag is removed and the connection is no longer throttled. Finally, the rate at which each flagged connection is throttled, i.e. the `FlagRate`, is statically defined and is not adjusted by the algorithm.

Note that the flagging parameters need only be set based on system-wide policy and generally do not require independent relay tuning, but provides the flexibility to allow individual relay operators to deviate from system policy if they desire.

Throttling Using Thresholds. Recall the two main issues a throttling algorithm must address: selecting *which connections* to throttle and the *rate* at which to throttle them. Our bit-splitting algorithm explored adaptively adjusting the throttle rate and applying this to all connections while our flagging algorithm explored statically configuring a throttle rate and adaptively selecting the throttled connections. We now describe our final algorithm which attempts to adaptively address both issues.

[4] The same is not true for the unthrottled connections between relays since each of them contain several circuits and each circuit may belong to a different client (see Section 2).

The threshold algorithm also makes use of a connection-level cell EWMA, which is computed as described above for the flagging algorithm. However, EWMA is used here to sort connections by the loudest to quietest. We then select and throttle the loudest fraction $\mathcal{T}$ of connections, where $\mathcal{T}$ is a configurable threshold. For example, setting $\mathcal{T}$ to 0.1 means the loudest ten percent of client connections will be throttled. The selection is adaptive since the EWMA changes over time according to each connection's bandwidth usage.

We have adaptively selected which connections to throttle and now must determine a throttle rate. To do this, we require that each connection tracks its throughput over time. We choose the average throughput rate of the connection with the minimum EWMA from the set of connections being throttled. For example, when $\mathcal{T}$ = 0.1 and there are 100 client connections sorted from loudest to quietest, the chosen throttle rate is the average throughput of the tenth connection. Each of first ten connections is then throttled at this rate. In our prototype, we approximate the throughput rate as the average number of bytes transferred over the last $\mathcal{R}$ seconds, where $\mathcal{R}$ is configurable. $\mathcal{R}$ represents the period of time between which the algorithm re-selects the throttled connections, adjusts the throttle rates, and resets each connection's throughput counters.

There is one caveat to the algorithm as described above. In our experiments in Section 4, we noticed that occasionally the throttle rate chosen by the threshold algorithm was zero. This would happen if the mean throughput of the threshold connection (line 6 in Algorithm 3) did not send data over the last $\mathcal{R}$ seconds. To prevent a throttle rate of zero, we added a parameter to statically configure a throttle rate floor $\mathcal{F}$ so that no connection would ever be throttled below $\mathcal{F}$. Algorithm 3 details threshold adaptive throttling.

4 Experiments

In this section we explore the performance benefits possible with each throttling algorithm specified in Section 3. We perform experiments with Shadow [2, 31], an accurate and efficient discrete event simulator that runs real Tor code over a simulated network. Shadow allows us to run an entire Tor network on a single machine and configure characteristics such as network latency, bandwidth, and topology. Since Shadow runs real Tor, it accurately characterizes application behavior and allows us to focus on experimental comparison of our algorithms. A direct comparison between Tor and Shadow-Tor performance is presented in [31].

Experimental Setup. Using Shadow, we configure a private Tor network with 200 `HTTP` servers, 950 Tor web clients, 50 Tor bulk clients, and 50 Tor relays. The distribution of clients in our experiments approximates that found by McCoy *et al.* [38]. All of our nodes run inside the Shadow simulation environment.

In our experiments, each client node runs Tor in client-only mode as well as an `HTTP` client application configured to download over Tor's `SOCKS` proxy available on the local interface. Each web client downloads a 320 KiB file[5] from a randomly selected one of our `HTTP` servers, and pauses for a length of time drawn from the UNC "think time" data set [27] before downloading the next file. Each bulk client repeatedly downloads a 5 MiB file from a randomly selected `HTTP` server without pausing. Clients track the time to the first and the last byte of the download as indications of network responsiveness and overall performance.

Tor relays are configured with bandwidth parameters according to a Tor network consensus document.[6] We configure our network topology and latency between nodes according to the geographical distribution of relays and pairwise PlanetLab node ping times. Our simulated network mirrors a previously published Tor network model [31] that has been compared to and shown to closely approximate the load of the live Tor network [3].

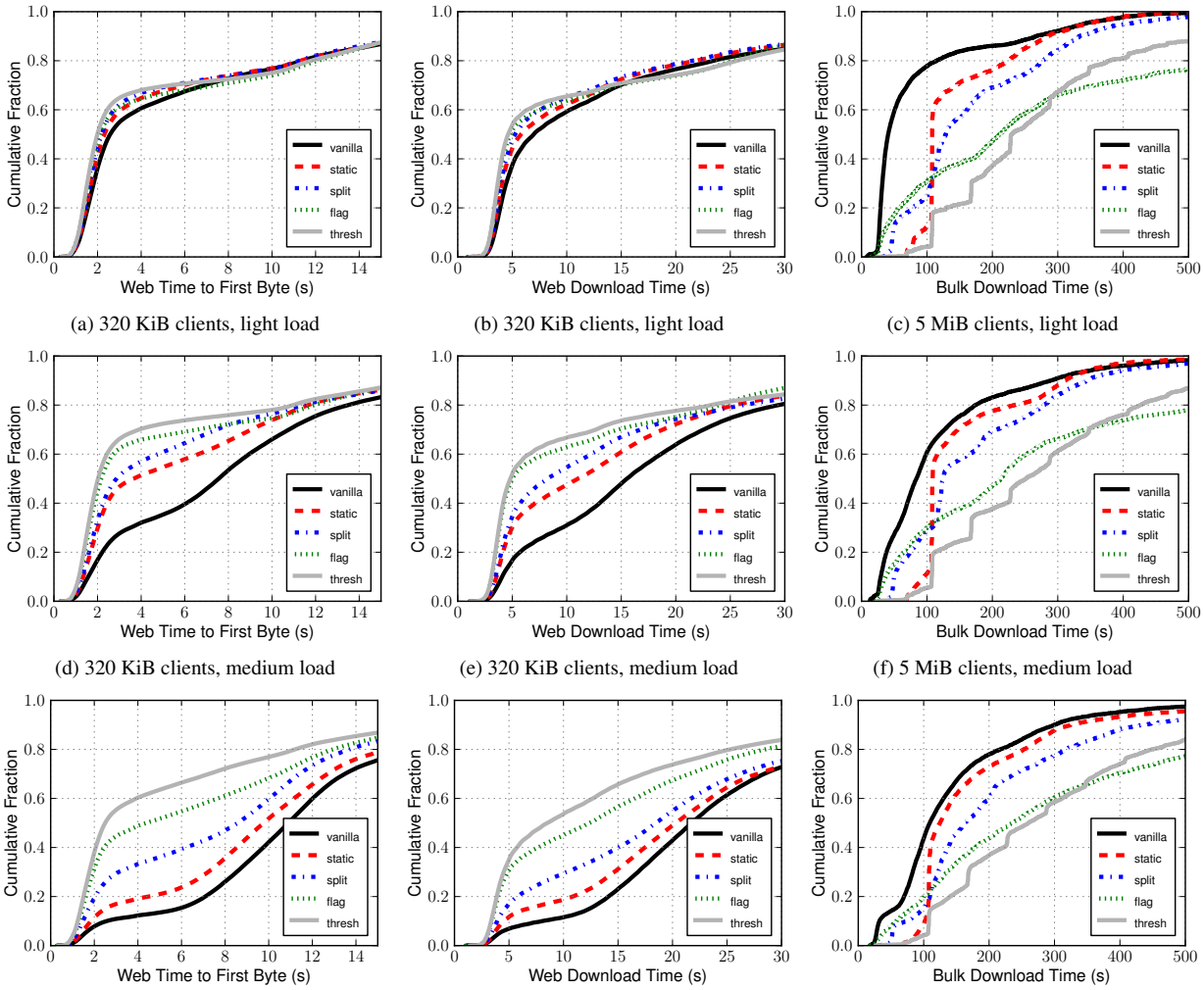
We focus on the time to the first data byte for web clients as a measure of network responsiveness, and the time to the last data byte—the download time—for both web and bulk clients as a measure of overall performance. In our results, "vanilla" represents unmodified Tor using a round-robin circuit scheduler and no throttling—the default settings in the Tor software—and can be used to compare relative performance between experiments. Each experiment uses network-wide deployments of each configuration. To further reduce random variances, we ran all configurations five times each. Therefore, every curve on every CDF shows the cumulative results of five experiments.

Results. Our results focus on the algorithmic configurations that we found to maximize web client performance [33] while we show how the algorithms perform when the network load varies from light (25 bulk clients) to medium (50 bulk clients) to heavy (100 bulk clients). The experimental setup is otherwise unmodified from the model described above. Running the algorithms under various loads allows us to highlight the unique and novel features each provides.

Figure 3 shows client performance for our algorithms. The time to first byte indicates network responsiveness for web clients while the download time indicates overall client performance for web and bulk clients. Client performance is shown for the lightly loaded network in Figures 3a–3c, the normally loaded network in Figures 3d–3f, and the heavily loaded network in Figures 3g–3i.

[5] The average webpage size reported by Google web metrics [45].

[6] Retrieved on 2011-04-27 and valid from 03-06:00:00



(a) 320 KiB clients, light load (b) 320 KiB clients, light load (c) 5 MiB clients, light load

(d) 320 KiB clients, medium load (e) 320 KiB clients, medium load (f) 5 MiB clients, medium load

(g) 320 KiB clients, heavy load (h) 320 KiB clients, heavy load (i) 5 MiB clients, heavy load

Figure 3: Comparison of client performance for each throttling algorithm and vanilla Tor, under various load. All experiments use 950 web clients. We vary the load between "light," "medium," and "heavy" by setting the number of bulk clients to 25 for 3a–3c, to 50 for 3d–3f, and to 100 for 3g–3i. The time to first byte indicates network responsiveness while the download time indicates overall client performance. The parameters for each algorithm are tuned based on experiments presented in [33].

Overall, static throttling results in the least amount of bulk traffic throttling while providing the lowest benefit to web clients. For the bit-splitting algorithm, we see improvements over static throttling for web clients for both time to first byte and overall download times, while download times for bulk clients are also slightly increased. Flagging and threshold throttling perform somewhat more aggressive throttling of bulk traffic and therefore also provide the greatest improvements in web client performance.

We find that each algorithm is effective at throttling bulk clients independent of network load, as evident in Figures 3c, 3f and 3i. However, performance benefits for web clients vary slightly as the network load changes. When the number of bulk clients is halved, throughput in Figure 3b is fairly similar across algorithms. However, when the number of bulk clients is doubled, responsiveness in Figure 3g and throughput in Figure 3h for both the static throttling and the adaptive bit-splitting algorithm lag behind the performance of the flagging and threshold algorithms. Static throttling would likely require a reconfiguration of throttling parameters while bit-splitting adjusts the throttle rate less effectively than our flagging and threshold algorithms.

As seen in Figures 3a, 3d, and 3g, as the load changes, the strengths of each algorithm become apparent. The flagging and threshold algorithms stand out as the best approaches for both web client responsiveness and throughput, and Figures 3c, 3f, and 3i show that they are also most aggressive at throttling bulk clients. The flagging algorithm appears very effective at accurately classifying bulk connections regardless of network

		vanilla	static	split	flag	thresh
light	Data (GiB)	88.3	80.3	78.3	72.1	69.8
	Web (%)	74.5	83.7	85.9	92.7	90.1
	Bulk (%)	25.5	16.3	14.1	7.3	9.9
medium	Data (GiB)	92.2	88.6	84.7	77.7	76.3
	Web (%)	65.8	72.4	75.0	86.2	82.8
	Bulk (%)	34.2	27.6	25.0	13.8	17.2
heavy	Data (GiB)	94.7	91.1	85.0	81.7	85.0
	Web (%)	55.8	60.5	64.3	75.4	71.2
	Bulk (%)	44.2	39.5	35.7	24.6	28.8

Table 1: Total data downloaded in our simulations by client type. Throttling reduces the bulk traffic share of the load on the network. The flagging algorithm is the best at throttling bulk traffic under light, medium, and heavy loads of 25, 50, and 100 bulk clients, respectively.

load. The threshold algorithm maximizes web client performance in our simulations among all loads and all algorithms tested, since it effectively throttles the worst bulk clients while utilizing extra bandwidth when possible. Both the threshold and flagging algorithms perform well over all network loads tested, and their usage in Tor would require little-to-no maintenance while providing significant performance improvements for web clients.

Aggregate download statistics are shown in Table 1. The results indicate that we are approximating the load distribution measured by McCoy *et al.* [38] reasonably well. The data also indicates that as the number of bulk clients in our simulation increases, so does the total amount of data downloaded and the bulk fraction of the total as expected. The data also shows that all throttling algorithms reduce the total network load. Static throttling reduces load the least, while our adaptive flagging algorithm is both the best at reducing both overall load and the bulk percentage of network traffic. Each of our adaptive algorithms are better at reducing load than static throttling, due to their ability to adapt to network dynamics. The relative difference between each algorithm's effectiveness at reducing load roughly corresponds to the relative difference in web client performance in our experiments, as we discussed above.

Discussion. The best algorithm for Tor depends on multiple factors. Although not maximizing web client performance, bit-splitting is the simplest, the most efficient, and the most network neutral approach (every connection is allowed the same portion of a guard's capacity). This "subtle" or "delicate" approach to throttling may be favorable if supporting multiple client behaviors is desirable. Conversly, the flagging algorithm may be used to identify a specific class of traffic and throttle it aggressively, creating the potential for the largest increase in performance for unthrottled traffic. We are currently exploring improvements to our statistical classification techniques to reduce false positives and to improve the control over traffic of various types. For these reasons, we feel the bit-splitting and flagging algorithms will be the most useful in various situations. We suggest that perhaps bit-splitting is the most appropriate throttling algorithm to use initially, even if something more aggressive is desirable in the long term.

While requiring little maintenance, our algorithms were designed to use only local relay information. Therefore, they are incrementally deployable while relay operators may choose the desired throttling algorithm independent of others. Our algorithms are already implemented in Tor and software patches are available [5].

5 Analysis and Discussion

Having shown the performance benefits of throttling bulk clients in Section 4, we now analyze the security of throttling against adversarial attacks on anonymity. We will discuss the direct impact of throttling on anonymity: what an adversary can learn when guards throttle clients and how the information leaked affects the anonymity of the system. We lastly discuss potential strategies clients may use to elude the throttles.

Before exploring practical attacks, we introduce two techniques an adversary may use to gather information about the network given that a generic throttling algorithm is enabled at all guards. Similar techniques used for throughput-based traffic analysis outside the context of throttling are discussed in detail by Mittal *et al.* [39]. Discussion about the security of our throttling algorithms in the context of practical attacks will follow.

5.1 Gathering Information

Our analysis uses the following terminology. At time t, the throughput of a connection between a client and a guard is λ_t, the rate at which the client will be throttled is α_t, and the allowed data burst is β. Note that, as consistent with our algorithms, the throttle rate may vary over time but the burst is a static system-wide parameter.

Probing Guards. Using the above terminology, a connection is throttled if, over the last s seconds, its throughput exceeds the allowed initial burst and the long-term throttle rate:

$$\sum_{k=t-s}^{t} (\lambda_k) \geq \beta + \sum_{k=t-s}^{t} (\alpha_k) \qquad (1)$$

A client may perform a simple technique to probe a specific guard node and determine the rate at which it gets throttled. The client may open a single circuit through the guard, selecting other high-bandwidth relays to ensure that the circuit does not contain a bottleneck. Then, it may download a large file and observe the change in throughput after receiving a burst of β payload bytes.

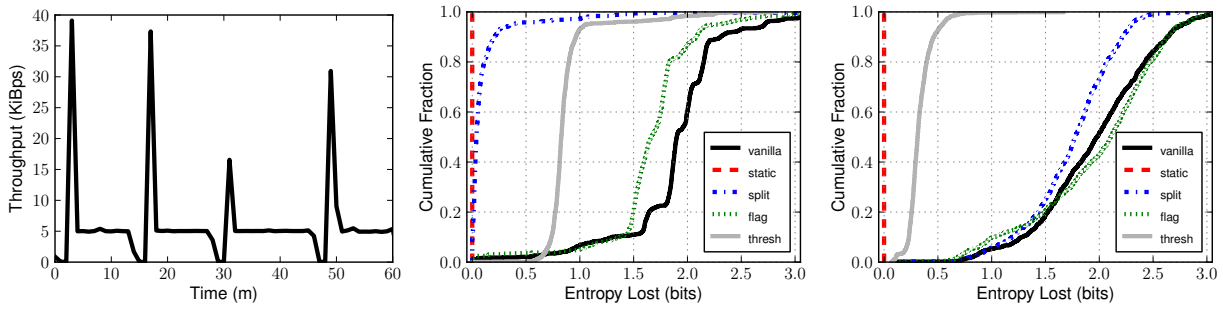


Figure 4: 4a: Client's may discover the throttle rate by probing guards. 4b: Information leaked by learning circuit throughputs. 4c: Information leaked by learning guards' throttle rates.

If the first β bytes are received at time t_1 and the download finishes at time $t_2 \geq t_1$, the throttle rate at any time t in this interval can be approximated by the mean throughput leading up to t:

$$\forall t \in [t_1, t_2],\ \alpha_t \approx \frac{\sum_{k=t_1}^{t}(\lambda_k)}{t - t_1} \tag{2}$$

Therefore, α_{t_2} approximates the actual throttle rate. Note that this approximation may under-estimate the actual throttle rate if the throughput falls below the throttle rate during the measured interval.

We simulate probing in Shadow [2, 31] to show its effectiveness against the static throttling algorithm. As apparent in Figure 4a, the throttle rate was configured at 5 KiB/s and the burst at 2 MiB. With enough resources, an adversary may probe every guard node to form a complete list of throttle rates.

Testing Circuit Throughput. A web server may determine the throughput of a connecting client's circuit by using a technique similar to that presented by Hopper *et al.* [30]. When the server gets an `HTTP` request from a client, it may inject either special JavaScript or a large amount of garbage `HTML` into a form element included in the response. The injected code will trigger a second client request after the original response is received. The server may adjust the amount of returned data and measure the time between when it sent the first response and when it received the second request to approximate the throughput of the circuit.

5.2 Adversarial Attacks

We now explore several adversarial attacks in the context of client throttling algorithms, and how an adversary may use those attacks to learn information and affect the anonymity of a client.

Attack 1. In our first attack, an adversary obtains a distribution on throttle rates by probing all Tor guard relays. We assume the adversary has resources to perform such an attack, e.g. by utilizing a botnet or other distributed network such as PlanetLab [13]. The adversary then obtains access to a web server and tests the throughput of a target circuit. With this information, the adversary may reduce the *anonymity set* of the circuit's potential guards by eliminating those whose throttle rate is inconsistent with the measured circuit throughput.

This attack is somewhat successful against all of the throttling algorithms we have described. For bit-splitting, the anonymity set of possible guard nodes will consist of those whose bandwidth and number of active connections would throttle to the throughput of the target circuit or higher. By running the attack repeatedly over time, an intersection will narrow the set to those whose throttle rate is consistent with the target circuit throughput at all measured times.

The flagging algorithm throttles all flagged connections to the same rate system-wide. (We assume here that the set of possible guards is already narrowed to those whose bandwidth is consistent with the target circuit's throughput irrespective of throttling.) A circuit whose throughput matches the system-wide rate is either flagged at some guard or just coincidentally matches the system-wide rate and is not flagged because its EWMA has remained below the `splitRate` (see Algorithm 2) for its guard long enough to not be flagged or become unflagged. The throttling rate is thus not nearly as informative as for bit-splitting. If we run the attack repeatedly however, we can eliminate from the anonymity set any guard such that the EWMA of the target circuit should have resulted in a throttling but did not. Also, if the EWMA drops to the throttling rate at precise times (ignoring unusual coincidence), we can eliminate any guard that would not have throttled at precisely those times. Note that this determination must be made after the fact to account for the burst bucket of the target circuit, but it can still be made precisely.

The potential for information going to the attacker in the threshold algorithm is a combination of the potential in each of the above two algorithms. The timing of when a circuit gets throttled (or does not when it should have been) can narrow the anonymity set of entry guards as in the flagging algorithm. Once the circuit has been throttled, then any fluctuation in the throttling rate that separates out the guard nodes can be used to further narrow the set. Note that if a circuit consistently falls below the throttling rate of all guards, an attacker can learn nothing about its possible entry guard from this attack. Attack 2 considerably improves the situation for the adversary.

We simulated this attack in Shadow [2, 31]. An adversary probes all guards and forms a distribution on the throttle rate at which a connection would become throttled. We then form a distribution on circuit throughputs over each minute, and remove any guard whose throttle rate is outside a range of one standard deviation of those throughputs. Since there are 50 guards, the maximum entropy is $\log_2(50) \approx 5.64$; the entropy lost by this attack for various throttling algorithms relative to vanilla Tor is shown in Figure 4b. We can see that the static algorithm actually loses no information, since all connections are throttled to the same rate, while vanilla Tor without throttling actually loses *more* information than any of the throttling algorithms. Therefore, the distribution on guard bandwidth leaks more information than throttled circuits' throughputs.

Attack 2. As in Attack 1, the adversary again obtains a distribution on throttle rates of all guards in the system. However, the adversary slightly modifies its circuit testing by continuously sending garbage responses. The adversary adjusts the size of each response so that it may compute the throughput of the circuit over time and approximates the rate at which the circuit is throttled. By comparing the estimated throttle rate to the distribution on guard throttle rates, the adversary may again reduce the anonymity set by removing guards whose throttle rate is inconsistent with the estimated circuit throttle rate.

For bit-splitting, by raising and lowering the rate of garbage sent, the attacker can match this with the throttled throughput of each guard. The only guards in the anonymity set would be those that share the same throttling rate that matches the flooded circuit's throughput at all times. To maximize what he can learn from flagging, the adversary should raise the EWMA of the target circuit at a rate that will allow him to maximally differentiate guards with respect to when they would begin to throttle a circuit. If this does not uniquely identify the guard, he can also use the rate at which he diminishes garbage traffic to try to learn more from when the target circuit stops being throttled. As in Attack 1 from the threshold algorithm, the adversary can match the signature of both fluctuations in throttling rate over time and the timing of when throttling is applied to narrow the set of possible guards for a target circuit.

We simulated this attack using the same data set as Attack 1. Figure 4c shows that a connection's throttle rate generally leaks slightly more information than its throughput. As in Attack 1, guards' bandwidth in our simulation leaks more information than the throttle rate of each connection for all but the flagging algorithm.

Attack 3. An adversary controlling two malicious servers can link streams of a client connecting to each of them at the same time. The adversary uses the circuit testing technique to send a response of $\frac{\beta}{2}$ bytes in size to each of two requests. Then, small "test" responses are returned after receiving the clients' second requests. If the throughput of each circuit when downloading the "test" response is consistently throttled, then it is possible that the requests are coming from the same client. This attack relies on the observation that all traffic on the same client-to-guard connection will be throttled at the same time since each connection has a single burst bucket.

This attack is intended to indicate if and when a circuit is throttled, rather than the throttling rate. It will therefore not be effective against bit splitting, but will work against flagging or threshold throttling.

Attack 4. Our final attack is an active denial of service attack that can be used to confirm a circuit's entry guard with high probability. In this attack, the adversary attempts to adjust the throttle rate of each guard in order to identify whether it carries a target circuit. An adversary in control of a malicious server may monitor the throughput of a target circuit over time, and may then open a large number of connections to each guard node until a decrease in the target circuit's throughput is observed. To confirm that a guard is on the target circuit, the adversary can alternate between opening and closing guard connections and continue to observe the throughput of the target circuit. If the throughput is consistent with the adversary's behavior, it has found the circuit's guard with high probability.

The one thing not controlled by the adversary in Attack 2 is a guard's criterion for throttling at a given time – `splitRate` for bit splitting and flagging and `selectIndex` for threshold throttling (see Algorithms 1, 2, and 3). All of these are controlled by the number of circuits at the guard, which Attack 4 places under the control of the adversary. Thus, under Attack 4, the adversary will have precise control over which circuits get throttled at which rate at all times and can therefore uniquely determine the entry guard.

Note that all of Attacks 1, 2, and 4 are intended to learn about the possible entry guards for an attacked circuit. Even if completely successful, this does not fully de-anonymize the circuit. But since guards themselves are chosen for persistent use by a client, they can add

to pseudonymous profiling and can be combined with other information, such as that uncovered by Attack 3, to either reduce anonymity of the client or build a richer pseudonymous profile of it.

5.3 Eluding Throttles

A client may try multiple strategies to avoid being throttled. A client may instrument its downloading application and the Tor software to send application data over multiple Tor circuits. However, these circuits will still be subject to throttling since each of them uses the same throttled TCP connection to the guard. A client may avoid this by attempting to create multiple TCP connections to the guard. In this case, the guard may easily recognize that the connection requests come from the same client and can either deny the establishment of multiple connections or aggregate the accounting of all connections to that client. A client may use multiple guard nodes and send application data over each separate guard connection, but the client significantly decreases its anonymity by subverting the guard mechanism [58, 59]. Finally, the client could run and use its own guard node and avoid throttling itself. Although this strategy may actually benefit the network since it reduces the amount of Tor's capacity consumed by the client, the cost of running a guard may be sufficient to prevent its wide-scale adoption.

Its important to note that the "cheating" techniques outlined above do not decrease the *security* or *performance* below what unthrottled Tor provides. At worst, even if all clients somehow manage to elude the throttles, performance and security both regress to that of unthrottled Tor. In other words, throttling can only *improve* the situation whether or not "cheating" occurs in practice.

6 Related Work

6.1 Improving Tor's Performance

Recent work on improving Tor's performance covers a wide range of topics, which we now enumerate.

Incentives. A recognition that Tor is limited by its bandwidth resources has resulted in several proposals for developing performance incentives for volunteering bandwidth as a Tor relay. New relays would provide additional resources and improve network performance. Ngan *et al.* explore giving better performance to relays that attain the `fast` and `stable` relay flags [43]. These relays are marked with a "gold star" in the directory. Gold star relays may build circuits through other gold star relays, improving download performance. This scheme has a severe anonymity problem: any relay on a gold star circuit can determine with absolute certainty that the client is also a gold star relay. Jansen *et al.* explore reducing anonymity problems from the gold star approach by distributing anonymous tickets to all clients [32]. Relays then collect tickets from clients in exchange for prioritized service and can prioritize their own traffic in return. However, a centralized bank limits the allowable number of tickets in circulation, leading to spending strategies that may reduce anonymity. Finally, Moore *et al.* independently explored using static throttling configurations as a way to produce incentives for users to run relays in Tortoise [41]. Tortoise's throttling configurations must be monitored as network load changes, and anonymity with Tortoise is slightly worse than with the gold star scheme: the intersection attack is improved since gold star nodes retain their gold stars for several months after dropping from the consensus, whereas Tortoise only unthrottles nodes that are in the current consensus.

Relay Selection. Snader and Borisov [51] suggest an algorithm where relays opportunistically measure their peers' performance, allowing clients to use empirical aggregations to select relays for their circuits. A user-tunable mechanism for selecting relays is built into the algorithm: clients may adjust how often the fast relays get chosen, trading off anonymity and performance while not significantly reducing either. It was shown that this approach increases accuracy of available bandwidth estimates and reduces reaction time to changes in network load while decreasing vulnerabilities to low-resource routing attacks. Wang *et al.* [57] propose a congestion-aware path selection algorithm where clients choose paths based on information gathered during opportunistic and active measurements of relays. Clients use latency as an indication of congestion, and reject congested relays when building circuits. Improvements were realized for a single client, but its unclear how the new strategy would affect the network if used by all clients.

Scheduling. Alternative scheduling approaches have recently gained interest. Tang and Goldberg [52] suggest each relay track the number of packets it schedules for each circuit. After a configurable time-period, packet counts are exponentially decayed so that data sent more recently has a greater influence on the packet count. For each scheduling decision, the relay flushes the circuit with the lowest cell count, favoring circuits that have not sent much data recently while preventing bursty traffic from significantly affecting scheduling priorities. Jansen *et al.* [32] investigate new schedulers based on the proportional differentiation model [21] and differentiable service classes. Relays track the delay of each service class and prioritize scheduling so that relative delays are proportional to configurable differentiation parameters, but the schedulers require a mecha-

nism (tickets) for differentiating traffic into classes. Finally, Tor's round-robin TCP read/write schedulers have recently been noted as a source of unfairness for relays that have an unbalanced number of circuits per TCP connection [54]. Tschorsch and Scheuermann suggest that a round-robin scheduler could approximate a max-min algorithm [24] by choosing among all circuits rather than all TCP connections. More work is required to determine the suitability of this approach in Tor.

Congestion. Improving performance and reducing congestion has been studied by taking an in-depth look at Tor's circuit and stream windows [7]. AlSabah *et al.* experiment with dynamically adjusting window sizes and find that smaller window sizes effectively reduce queuing delays, but also decrease bandwidth utilization and therefore hurt overall download performance. As a result, they implement and test an algorithm from ATM networks called the N23 scheme, a link-by-link flow control algorithm. Their adaptive N23 algorithm propagates information about the available queue space to the next upstream router while dynamically adjusting the maximum circuit queue size based on outgoing cell buffer delays, leading to a quicker reaction to congestion. Their experiments indicate slightly improved response and download times for 300 KiB files.

Transport. Tor's performance has also been analyzed at the socket level, resulting in suggestions for a UDP-based mechanism for data delivery [56] or using a user-level TCP stack over a DTLS tunnel [47]. While Tor currently multiplexes all circuits over a single kernel TCP stream to control information leakage, the TCP-over-DTLS approach suggests separate user TCP streams for each circuit and sends all TCP streams between two relays over a single kernel DTLS-secured [40] UDP socket. As a result, a circuit's TCP window is not unfairly reduced when other high-bandwidth circuits cause queuing delays or dropped packets.

6.2 Bandwidth Management

Our approach to bandwidth management in this paper has been to use a token bucket rate-limiter, a classic traffic shaping mechanism [55], to ensure that traffic conforms to the desired policies. We now briefly discuss other approaches to bandwidth management.

Quality of Service. Networks often want to provide a certain quality of service (QoS) to their subscribers. There are two main approaches to QoS: Integrated Services (IntServ) and Differentiated Services (DiffServ).

In the IntServ [11, 50] model, applications request resources from the network using the resource reservation protocol [60]. Since the network must maintain the expected quality for its current commitments, it must ensure the load of the network remains below a certain level. Therefore, new requests may be denied if the network is unable to provide the resources requested. This approach does not work well in an anonymity network like Tor since clients would be able to request unbounded resources without accountability and the network would be unable to fulfill most requests due to bottlenecks.

In the DiffServ [9] model, applications notify the network of the desired service type by setting bits in the IP header. Routers then tailor performance toward an expected notion of fairness (e.g. max-min fairness [24, 34] or proportional fairness [20, 21, 35]). Leaking this type of information about a client's traffic flows is a significant risk to privacy and ways to provide differentiated service without such risk do not currently exist.

Scheduling. Scheduling algorithms, such as fair queuing [15] and round robin [24, 25], affect the order in which packets are sent out of a given node, but generally do not change the total number of packets being sent. Therefore, unless the sending rate is explicitly reduced, the network will still contain similar load regardless of the relative priority of individual packets. As explained in Section 1 and Section 3, scheduling does not directly reduce network congestion, but may cooperate with other bandwidth management techniques to achieve the desired performance characteristics of traffic classes.

7 Conclusion

This paper analyzes client throttling by guard relays to reduce Tor network bottlenecks and improve responsiveness. We explore static throttling configurations while designing, implementing, and evaluating three new throttling algorithms that adaptively select which connections get throttled and dynamically adjust the throttle rate of each connection. Our adaptive throttling techniques use only local relay information and are considerably more effective than static throttling since they do not require re-evaluation of throttling parameters as network load changes. We find that client throttling is effective at both improving performance for interactive clients and increasing Tor's network resilience. We also analyzed the effects throttling has on anonymity and discussed the security of our algorithms against realistic adversarial attacks. We find that throttling improves anonymity: a guard's bandwidth leaks more information about its circuits when throttling is *disabled*.

Future Work. There are many directions for future research. Our current algorithms may be modified to optimize performance by improving classification of bulk traffic, considering alternative strategies for distinguishing web from bulk connections. Additional approaches to rate-tuning are also of interest, e.g. it may be possible to further improve web client performance using proportional fairness to schedule traffic on circuits. Also of

interest is an analysis of throttling in the context of congestion and flow control to determine the interrelation and effects the algorithms have on each other. Finally, a deeper understanding of our algorithms and their effects on client performance would be possible through analysis on the live Tor network.

Acknowledgements. We thank Roger Dingledine for helpful discussions regarding this work and the anonymous reviewers for their feedback and suggestions. This research was supported by NFS grant CNS-0917154, ONR, and DARPA.

References

[1] The Libevent Event Notification Library, Version 2.0. http://monkey.org/~provos/libevent/.

[2] The Shadow Simulator. http://shadow.cs.umn.edu/.

[3] The Tor Metrics Portal. http://metrics.torproject.org/.

[4] The Tor Project. https://www.torproject.org/.

[5] Throttling Algorithms Code Repository. https://github.com/robgjansen/torclone.

[6] ACQUISTI, A., DINGLEDINE, R., AND SYVERSON, P. On the Economics of Anonymity. In *Proceedings of Financial Cryptography* (January 2003), R. N. Wright, Ed., Springer-Verlag, LNCS 2742.

[7] ALSABAH, M., BAUER, K., GOLDBERG, I., GRUNWALD, D., MCCOY, D., SAVAGE, S., AND VOELKER, G. DefenestraTor: Throwing out Windows in Tor. In *Proceedings of the 11th International Symposium on Privacy Enhancing Technologies* (2011).

[8] BACK, A., MOLLER, U., AND STIGLIC, A. Traffic Analysis Attacks and Trade-offs in Anonymity Providing Systems. In *Proceedings of Information Hiding Workshop* (2001), pp. 245–257.

[9] BLAKE, S., BLACK, D., CARLSON, M., DAVIES, E., WANG, Z., AND WEISS, W. An Architecture for Differentiated Services, 1998.

[10] BORISOV, N., DANEZIS, G., MITTAL, P., AND TABRIZ, P. Denial of Service or Denial of Security? In *Proceedings of the 14th ACM conference on Computer and communications security* (2007), ACM, pp. 92–102.

[11] BRADEN, B., CLARK, D., AND SHENKER, S. Integrated Service in the Internet Architecture: an Overview.

[12] CHEN, F., AND PERRY, M. Improving Tor Path Selection. https://gitweb.torproject.org/torspec.git/blob/HEAD:/proposals/151-path-selection-improvements.txt.

[13] CHUN, B., CULLER, D., ROSCOE, T., BAVIER, A., PETERSON, L., WAWRZONIAK, M., AND BOWMAN, M. PlanetLab: an Overlay Testbed for Broad-coverage Services. *SIGCOMM Computer Communication Review 33* (2003), 3–12.

[14] CROTTI, M., DUSI, M., GRINGOLI, F., AND SALGARELLI, L. Traffic Classification Through Simple Statistical Fingerprinting. *SIGCOMM Comput. Commun. Rev. 37* (January 2007), 5–16.

[15] DEMERS, A., KESHAV, S., AND SHENKER, S. Analysis and Simulation of a Fair Queueing Algorithm. In *ACM SIGCOMM Computer Communication Review* (1989), vol. 19, ACM, pp. 1–12.

[16] DINGLEDINE, R. Iran Blocks Tor. https://blog.torproject.org/blog/iran-blocks-tor-tor-releases-same-day-fix.

[17] DINGLEDINE, R. Research problem: adaptive throttling of Tor clients by entry guards. https://blog.torproject.org/blog/research-problem-adaptive-throttling-tor-clients-entry-guards.

[18] DINGLEDINE, R., AND MATHEWSON, N. Anonymity Loves Company: Usability and the Network Effect. In *Proceedings of the Fifth Workshop on the Economics of Information Security (WEIS 2006), Cambridge, UK, June* (2006).

[19] DINGLEDINE, R., MATHEWSON, N., AND SYVERSON, P. Tor: The Second-Generation Onion Router. In *Proceedings of the 13th USENIX Security Symposium* (2004).

[20] DOVROLIS, C., AND RAMANATHAN, P. A Case for Relative Differentiated Services and the Proportional Differentiation Model. *Network, IEEE 13*, 5 (1999), 26–34.

[21] DOVROLIS, C., STILIADIS, D., AND RAMANATHAN, P. Proportional Differentiated Services: Delay Differentiation and Packet Scheduling. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication* (1999), pp. 109–120.

[22] EVANS, N., DINGLEDINE, R., AND GROTHOFF, C. A Practical Congestion Attack on Tor Using Long Paths. In *Proceedings of the 18th USENIX Security Symposium* (2009), pp. 33–50.

[23] GOLDSCHLAG, D. M., REED, M. G., AND SYVERSON, P. F. Hiding Routing Information. In *Proceedings of Information Hiding Workshop* (1996), pp. 137–150.

[24] HAHNE, E. Round-robin Scheduling for Max-min Fairness in Data Networks. *IEEE Journal on Selected Areas in Communications 9*, 7 (1991), 1024–1039.

[25] HAHNE, E., AND GALLAGER, R. Round-robin Scheduling for Fair Flow Control in Data Communication Networks. *NASA STI/Recon Technical Report N 86* (1986), 30047.

[26] HARDIN, G. The Tragedy of the Commons. *Science 162*, 3859 (December 1968), 1243–1248.

[27] HERNANDEZ-CAMPOS, F., JEFFAY, K., AND SMITH, F. Tracking the Evolution of Web Traffic: 1995-2003. In *The 11th IEEE/ACM International Symposium on Modeling, Analysis, and Simulation of Computer Telecommunications Systems* (2003), pp. 16–25.

[28] HINTZ, A. Fingerprinting Websites using Traffic Analysis. In *Proceedings of Privacy Enhancing Technologies Workshop* (2002), pp. 171–178.

[29] HJELMVIK, E., AND JOHN, W. Statistical Protocol Identification with SPID: Preliminary Results. In *Swedish National Computer Networking Workshop* (2009).

[30] HOPPER, N., VASSERMAN, E., AND CHAN-TIN, E. How Much Anonymity Does Network Latency Leak? *ACM Transactions on Information and System Security 13*, 2 (2010), 1–28.

[31] JANSEN, R., AND HOPPER, N. Shadow: Running Tor in a Box for Accurate and Efficient Experimentation. In *Proceedings of the 19th Network and Distributed System Security Symposium* (2012).

[32] JANSEN, R., HOPPER, N., AND KIM, Y. Recruiting New Tor Relays with BRAIDS. In *Proceedings of the 17th ACM Conference on Computer and Communications Security* (2010), pp. 319–328.

[33] JANSEN, R., SYVERSON, P., AND HOPPER, N. Throttling Tor Bandwidth Parasites. Tech. Rep. 11-019, University of Minnesota, 2011.

[34] KATEVENIS, M. Fast Switching and Fair Control of Congested Flow in Broadband Networks. *Selected Areas in Communications, IEEE Journal on 5*, 8 (1987), 1315–1326.

[35] KELLY, F., MAULLOO, A., AND TAN, D. Rate Control for Communication Networks: Shadow Prices, Proportional Fairness and Stability. *Journal of the Operational Research society 49*, 3 (1998), 237–252.

[36] KOHNEN, C., UBERALL, C., ADAMSKY, F., RAKOCEVIC, V., RAJARAJAN, M., AND JAGER, R. Enhancements to Statistical Protocol IDentification (SPID) for Self-Organised QoS in LANs. In *Computer Communications and Networks (ICCCN), 2010 Proceedings of 19th International Conference on* (2010), IEEE, pp. 1–6.

[37] LOESING, K. Measuring the Tor network: Evaluation of client requests to directories. Tech. rep., Tor Project, 2009.

[38] MCCOY, D., BAUER, K., GRUNWALD, D., KOHNO, T., AND SICKER, D. Shining Light in Dark Places: Understanding the Tor Network. In *Proceedings of the 8th International Symposium on Privacy Enhancing Technologies* (2008), pp. 63–76.

[39] MITTAL, P., KHURSHID, A., JUEN, J., CAESAR, M., AND BORISOV, N. Stealthy Traffic Analysis of Low-Latency Anonymous Communication Using Throughput Fingerprinting. In *Proceedings of the 18th ACM conference on Computer and Communications Security* (October 2011).

[40] MODADUGU, N., AND RESCORLA, E. The Design and Implementation of Datagram TLS. In *Proceedings of the 11th Network and Distributed System Security Symposium* (2004).

[41] MOORE, W. B., WACEK, C., AND SHERR, M. Exploring the Potential Benefits of Expanded Rate Limiting in Tor: Slow and Steady Wins the Race With Tortoise. In *Proceedings of 2011 Annual Computer Security Applications Conference* (December 2011).

[42] MURDOCH, S., AND DANEZIS, G. Low-cost Traffic Analysis of Tor. In *IEEE Symposium on Security and Privacy* (2005), pp. 183–195.

[43] NGAN, T.-W. J., DINGLEDINE, R., AND WALLACH, D. S. Building Incentives into Tor. In *The Proceedings of Financial Cryptography* (2010).

[44] ØVERLIER, L., AND SYVERSON, P. Locating Hidden Servers. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy* (May 2006), IEEE CS.

[45] RAMACHANDRAN, S. Web Metrics: Size and Number of Resources. http://code.google.com/speed/articles/web-metrics.html, 2010. Accessed February, 2012.

[46] RAYMOND, J. Traffic Analysis: Protocols, Attacks, Design Issues, and Open Problems. In *Designing Privacy Enhancing Technologies* (2001), pp. 10–29.

[47] REARDON, J., AND GOLDBERG, I. Improving Tor using a TCP-over-DTLS tunnel. In *Proceedings of the 18th USENIX Security Symposium* (2009).

[48] REED, M., SYVERSON, P., AND GOLDSCHLAG, D. Anonymous Connections and Onion Routing. *IEEE Journal on Selected Areas in Communications 16*, 4 (1998), 482–494.

[49] SERJANTOV, A., AND SEWELL, P. Passive Attack Analysis for Connection-based Anonymity Systems. *Computer Security–ESORICS* (2003), 116–131.

[50] SHENKER, S., PARTRIDGE, C., AND GUERIN, R. RFC 2212: Specification of Guaranteed Quality of Service, Sept. 1997. Status: PROPOSED STANDARD.

[51] SNADER, R., AND BORISOV, N. A Tune-up for Tor: Improving Security and Performance in the Tor Network. In *Proceedings of the 16th Network and Distributed Security Symposium* (2008).

[52] TANG, C., AND GOLDBERG, I. An Improved Algorithm for Tor Circuit Scheduling. In *Proceedings of the 17th ACM Conference on Computer and Communications Security* (2010), pp. 329–339.

[53] TSCHORSCH, F., AND SCHEUERMANN, B. Refill Intervals. https://gitweb.torproject.org/torspec.git/blob/HEAD:/proposals/183-refillintervals.txt.

[54] TSCHORSCH, F., AND SCHEUERMANN, B. Tor is Unfair–and What to Do About It, 2011.

[55] TURNER, J. New directions in communications(or which way to the information age?). *IEEE communications Magazine 24*, 10 (1986), 8–15.

[56] VIECCO, C. UDP-OR: A Fair Onion Transport Design. In *Proceedings of Hot Topics in Privacy Enhancing Technologies* (2008).

[57] WANG, T., BAUER, K., FORERO, C., AND GOLDBERG, I. Congestion-aware Path Selection for Tor. In *Proceedings of Financial Cryptography* (2012).

[58] WRIGHT, M., ADLER, M., LEVINE, B. N., AND SHIELDS, C. An Analysis of the Degradation of Anonymous Protocols. In *Proceedings of the Network and Distributed Security Symposium - NDSS '02* (February 2002), IEEE.

[59] WRIGHT, M., ADLER, M., LEVINE, B. N., AND SHIELDS, C. Defending Anonymous Communication Against Passive Logging Attacks. In *Proceedings of the 2003 IEEE Symposium on Security and Privacy* (May 2003), pp. 28–43.

[60] ZHANG, L., DEERING, S., ESTRIN, D., SHENKER, S., AND ZAPPALA, D. Rsvp: A new resource reservation protocol. *Network, IEEE 7*, 5 (1993), 8–18.