Technical Report

Department of Computer Science
and Engineering
University of Minnesota
4-192 Keller Hall
200 Union Street SE
Minneapolis, MN 55455-0159 USA

TR 11-019

Throttling Tor Bandwidth Parasites

Rob Jansen, Paul Syverson, and Nicholas J. Hopper

September 23, 2011

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's network congestion and performance problems stem from a small percentage of users that consume a large fraction of available relay bandwidth. These users continuously drain relays of excess bandwidth, creating new network bottlenecks and exacerbating the effects of existing ones. Attacking the problem at its source, we present the design of three new algorithms that throttle clients to reduce network congestion and increase interactive client performance. Unlike existing techniques, our algorithms adaptively adjust throttling parameters given only information local to a relay. We implement our algorithms in Tor and compare significant client performance benefits using network-wide deployments of our algorithms under a variety of network loads. We also analyze the effects of throttling on anonymity and compare the security of our algorithms under adversarial attack. Software patches for our algorithms will be submitted to Tor.

1 Introduction

The Tor [5] anonymity network was developed in an attempt to improve anonymity on the Internet. Onion Routing [8, 23] serves as the cornerstone for Tor's overlay network design. Tor *clients* source and encrypt messages in several "layers", packaging them into 512-byte packets called *cells* and sending 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. Traffic and timing analysis is possible [2, 7, 12, 13, 19, 21, 24], but complicated since each relay simultaneously services multiple circuits.

Tor relays are run by volunteers located throughout the world and service hundreds of thousands of Tor clients [16] with high bandwidth demands. A relay's utility to Tor is dependent on both the bandwidth *capabilities* 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 KiBps and become bottlenecks when chosen for a circuit. Bandwidth bottlenecks lead to network congestion and impair client performance. Moreover, bottlenecks are aggravated by bulk users, which make up roughly five percent of connections and forty percent of the bytes transferred through the network [17]. Bulk traffic further increases network-wide congestion and punishes interactive users as they attempt to browse the web and run `SSH` sessions, turning many away from Tor. As a result, both Tor's client diversity and anonymity suffer.

Shifting Bits. To reduce bottlenecks and improve performance, 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 that take too long to build [3]. This removes the worst of bottlenecks and improves usability, however, it does not solve all congestion problems. Low bandwidth relays must still be chosen for circuits to avoid significant anonymity problems, meaning there are

*Work partially performed at the University of Minnesota and the U.S. Naval Research Laboratory

still a large number of circuits with tight bandwidth bottlenecks. Another approach is to change the circuit scheduler to prioritize bursty (i.e. web) traffic over bulk traffic using an exponetially-weighted moving average (EWMA) of relayed cells [28]. Early experiments show small improvements at a single relay, but it is unclear how performance is affected when deployed to the entire network. These approaches shift network load to better utilize the available bandwidth, but in general do not increase the capacity of the network.

New Capabilities. Another approach to reducing congestion 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 [14, 20]. These approaches are interesting, however, they have not been deployed due to technical reasons and a lack of understanding of the economic implications they would impose on Tor and its users. It is unclear how an incentive scheme will affect users' motivation to contribute: 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 [31] and other organizations. While effective at improving network performance, 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 who wish to stay anonymous for legal reasons. The result is a classic *Tragedy of the Commons* [10] scenario: the bulk users attracted to the faster network will continue to leech the additional bandwidth.

Throttling to Lighten Load. Tor has recently built mechanisms into the software to throttle client connections [30]. Throttling will prevent bulk clients from consuming too much of the available bandwidth by intentionally imposing *restrictions* on clients' throughput. By reducing the throughput of bulk clients, we effectively reduce the rate at which new bulk requests are made, resulting in fewer bottlenecks and a less congested and more responsive Tor network. The throttling parameters are configurable but static, although Tor does not enable throttling by default.

We investigate throttling client connections in Section 3. We introduce and test three new adaptive throttling algorithms that use information local to a relay to dynamically select which connections get throttled and adjust the rate at which those connections are throttled. Our first algorithm adaptively adjusts the throttle rate, our second adaptively selects which connections get throttled, and our third adaptively adjusts both the throttle rate and the selected connections. We implement our algorithms in Tor[1] and test their effectiveness at improving performance in large scale, network-wide deployments. We test various configurations of our algorithms in Section 4 and compare our results to static throttling under a variety of network loads. We find that the effectiveness of static throttling configurations are highly dependent on network load whereas our adaptive algorithms work well under various load with no configuration changes. We find our algorithms impossible to misconfigure in a way that reduces client performance. We conclude that throttling is an effective approach to increasing performance for web clients.

Having shown that throttling provides significant performance benefits for web clients, we perform the first in-depth analysis of the anonymity implications of throttling while analyzing the security of our algorithms under adversarial attack. We discuss attacks on throttling in Section 5 and compare information leakage among our algorithms, finding that more anonymity is lost when throttling is *disabled*.

2 Background

Multiplexed Connections. We now explore the architecture of a Tor relay, shown in Figure 1, to facilitate an understanding of delays in the Tor network. 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. Further, a user may be simultaneously accessing multiple

[1]Software patches for our algorithms will be released to the Tor developers.

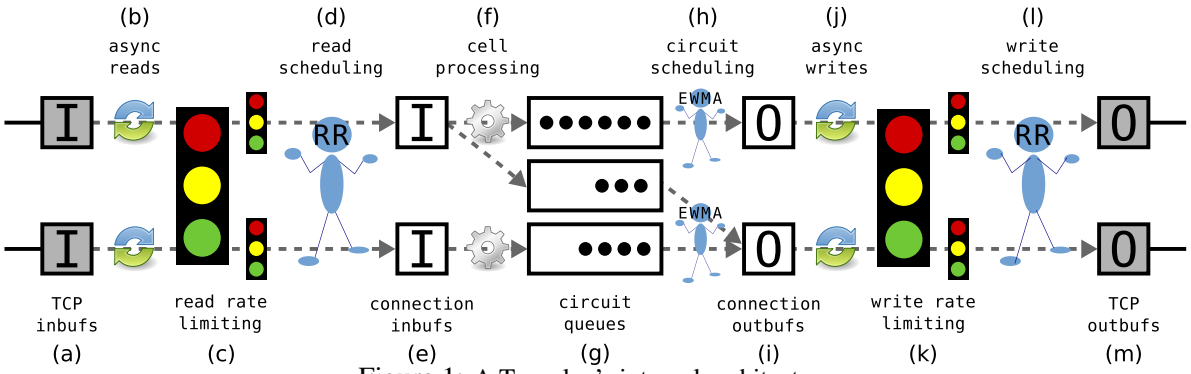


Figure 1: A Tor relay's internal architecture.

services or *streams*, each of which has it own TCP connection to the Tor client and may be multiplexed over one or more circuits. TCP offers reliability, in-order delivery of packets between relays, and potentially unfair kernel-level congestion control when multiplexing connections [22]. The distinction between and interaction of connections, circuits, and streams is important for understanding Tor's internals.

Connection Input. Tor uses libevent [15] to handle input and output from and to kernel TCP buffers. Tor registers sockets in which it is interested in reading 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 through the use of 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. 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 not read and cancels the callback 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 like bulk transfers and generally leads to better performance for others [30].

When a TCP input buffer is eligible for reading, a round-robin (RR) scheduling mechanism is used to read at most $\frac{1}{8}$ 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 [32]. 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 the TCP connection input buffer associated with a stream when the stream window reaches zero, and stops reading from all connections 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 `SENDME`s 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 `SENDME`s 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 [1].

Cell Processing and Queuing. Data is immediately processed as it arrives in connection input buffers (Figure 1f). First, a layer of encryption is either added or removed from each cell depending on the direction

it is traveling through the circuit. The cell is then switched onto the circuit corresponding to the next hop and placed into the circuit's unbounded 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 [28] 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 at most $\frac{1}{8}$ 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 [17]. The overwhelming load these clients place on the network increases congestion and creates additional bottlenecks, causing interactive traffic like instant messaging and `SSH` sessions to lose responsiveness. 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.

3.1 Static Throttling

Recently, Tor introduced the ability for entry guards to throttle connections to clients [30]. 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 ensures that all 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 only connections between guards and clients are throttled, and Tor does not enable client throttling by default.

3.2 Adaptive Throttling

Static throttling may allow each guard to prevent clients from overloading the network. However, as we will show in Section 4, the correct settings for the static algorithm depend on the load and current state of the network. It is possible to misconfigure the static algorithm and reduce performance for all clients. Therefore, 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 on 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 connection at the *statically* specified rate. Each of our three algorithms below answer these questions *adaptively* by considering information local to each relay. Note that our algorithms adaptively adjust the `PerConnBWRate` while keeping a constant `PerConnBWBurst`.

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 fair share of bandwidth unless it has unused tokens in its burst bucket. This approach will prevent bulk clients from unfairly consuming bandwidth and ensure that there is always bandwidth available for web clients. As shown in Algorithm 1, our bit-splitting algorithm is simple to implement, does not require relays to track any additional information, and requires no configuration or maintenence of parameters.

Algorithm 1 Throttling clients by splitting bits.

1: $B \leftarrow getRelayBandwidth()$
2: $L \leftarrow getConnectionList()$
3: $N \leftarrow L.length()$
4: **if** $N > 0$ **then**
5: $\quad splitRate \leftarrow \frac{B}{N}$
6: $\quad$ **for** $i \leftarrow 1$ to N **do**
7: $\quad\quad$ **if** $L[i].isClientConnection()$ **then**
8: $\quad\quad\quad L[i].throttleRate \leftarrow splitRate$
9: $\quad\quad$ **end if**
10: $\quad$ **end for**
11: **end if**

Flagging Unfair Clients. Our previous 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. Our throttling algorithm specifically attempts to avoid throttling web clients.

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

The flagging algorithm, shown in Algorithm 2, requires that each guard keeps an exponentially-weighted moving average (EWMA) of the number of cells that have been recently sent on each client connection. The per-connection cell EWMA is computed in much the same way as the per-circuit cell EWMA introduced by Tang and Goldberg [28]: whenever the circuit's cell counter is incremented, so is the cell counter of the outgoing connection to which that circuit belongs. Note that connection-level cell EWMA is fair in the sense that clients can not affect each other's EWMA since all circuits multiplexed over this guard-to-client connection belong to a single client. The same is not true for connections between relays since each of them contain several circuits and each circuit may belong to a different client

(see Section 2). The per-connection EWMA may be enabled independently of its circuit counterpart, and is configured using a separate half-life.

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 the connection was 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 independently 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 [28]. 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 to recover and stop throttling incorrectly flagged connections. 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 it 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. Algorithm 2 details the flagging approach.

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.

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 at each relay.

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

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 connection 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 our threshold approach to adaptive throttling.

4 Experiments

In this section we explore the performance benefits possible with each throttling algorithm specified in Section 3. Our experiments are performed with Shadow [25, 26], 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, configuring characteristics such as network latency, bandwidth, and topology. Since Shadow runs real Tor, it accurately characterizes application behavior while our algorithms implemented in Tor are simple to test and compare.

Using Shadow, we configure a Tor network with 200 `HTTP` servers, 950 Tor web clients, 50 Tor bulk clients, and 50 Tor relays. Each web client downloads a 320 KiB file from a randomly selected web server, and pauses for a length of time drawn from the UNC "think time" dataset [11] before downloading the next file. Each bulk client repeatedly downloads a 5 MiB file from a randomly selected web server without pausing. Each client tracks the time to the first byte of the download and the overall download time as indications of network responsiveness and overall performance. The client distribution approximates that found by McCoy *et al.* [17]. Tor relays are configured with bandwidth parameters according to a recently retrieved consensus document[2]. We configure our network topology and latency between nodes according to the geographical distribution of relays and PlanetLab ping times [26]. Our simulated network approximates the load of the live Tor network [29].

We experiment with various algorithmic parameters under identical load to determine the effect throttling has on web and bulk client performance. In all of our graphs throughout this section, "vanilla" represents unmodified Tor using a round-robin circuit scheduler and no throttling and can be used to compare results between different experiments. Each experiment uses network-wide deployments of each configuration. To reduce random variances, we run all experimental configurations five times each. Therefore, each curve on the CDFs throughout this section show the cumulative results of five experiments.

4.1 Static Throttling

We first test statically throttling clients using our experimental setup from above. Figures 2a–2d show client performance with various `PerConnBWRate` configurations, as shown in the legend, while configuring a constant 2 MiB `PerConnBWBurst`. Although not shown in Figure 2d, a rate of 5 KiBps significantly reduces performance for bulk connections. Bulk downloads of 5 MiB files take between 600 and 1300 seconds to complete, since the first 2 MiB of a new connection are unthrottled while the remaining 3 MiB, and the entire 5 MiB of subsequent downloads, are throttled at 5 KiBps. Unfortunately, as shown in Figure 2b, performance for web clients also decreases for those who download enough to exceed the allowed 2 MiB burst over the lifetime of a circuit. As we increase the throttle rate to 50 KiBps, we see that most bulk downloads take just over 100 seconds to complete. Again, the first download on a new connection completes somewhat faster because only 3 of the 5 MiB file is throttled. We also see improved time to first byte and improved download times for web clients as a direct result of throttling bulk connections. Increasing

[2]The consensus was retrieved on 2011-04-27 and valid between 03:00:00 and 06:00:00.

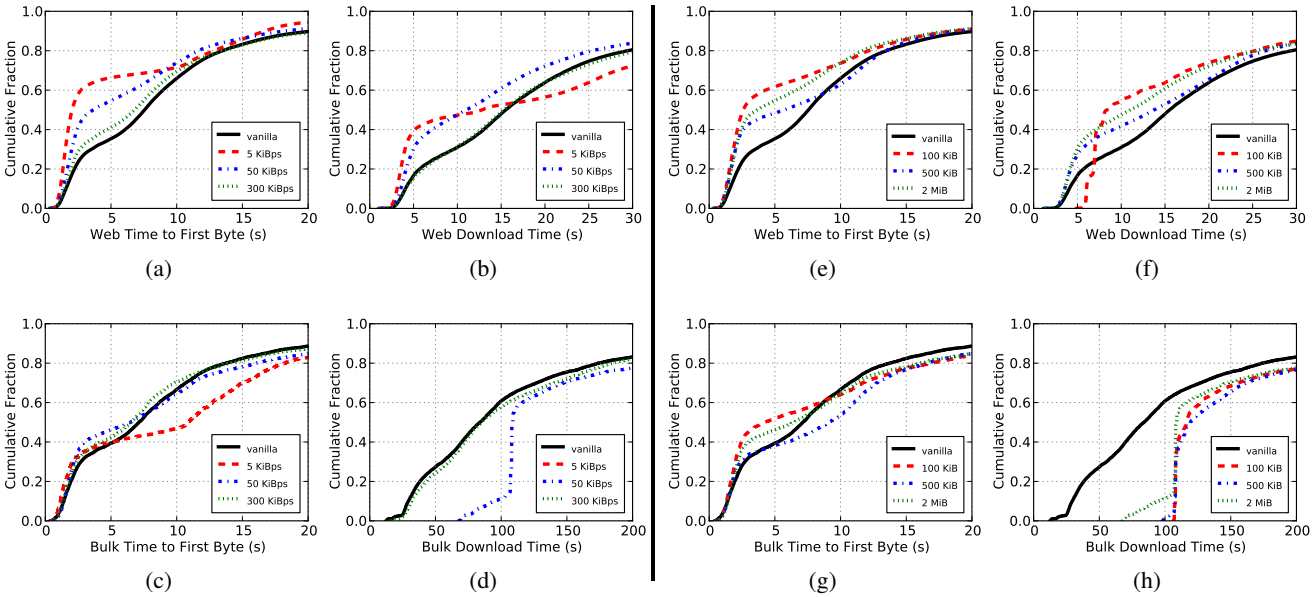


Figure 2: Throttling clients with static throttle configurations. (a)–(d) show performance with a varied `PerConnBWRate` as shown and a constant 2 MiB `PerConnBWBurst` while (e)–(h) use a constant 50 KiBps `PerConnBWRate` and varied `PerConnBWBurst` as shown. A rate of 50 KiBps and burst of 2 MiB leads to the most significant performance improvements in our simulations.

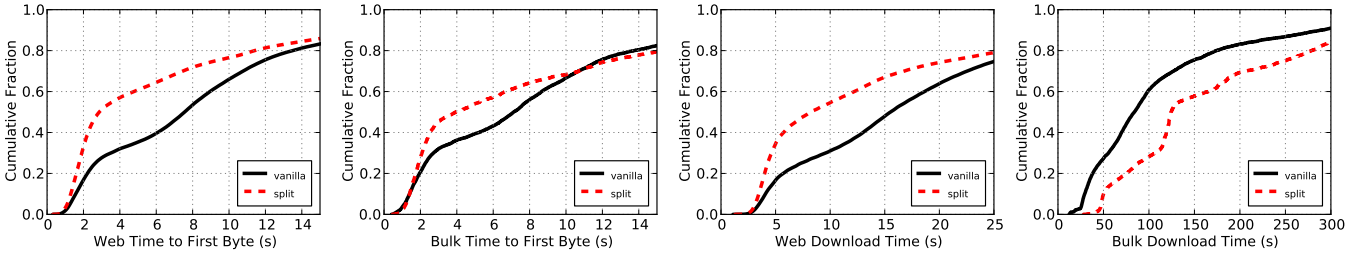
the throttle rate to 300 KiBps practically eliminates throttling of both web and bulk clients. Results for 300 KiBps and higher throttle rates converge with the results obtained with vanilla Tor, as expected.

Figures 2e–2h show client performance with a constant 50 KiBps `PerConnBWRate` and various `PerConnBWBurst` configurations, as shown in the legend. We again find that if we configure the burst to be too small, then performance will be reduced for both web and bulk clients because they will both exceed the burst and be throttled. As we increase the burst, then more web clients are never throttled because they do not exceed the burst. Their performance improves at the expense of the bulk clients. Our experiments show that throttling can be effective at improving performance for interactive clients but can be challenging to properly configure. We suspect that appropriate static throttling configurations (50 KiBps rate and 2 MiB burst in our simulations) will change as the load on the network changes. Therefore, we stress the importance of frequently assessing the network when statically throttling not only to optimize interactive client performance, but also to ensure the selected configuration is not hurting performance.

4.2 Adaptive Throttling

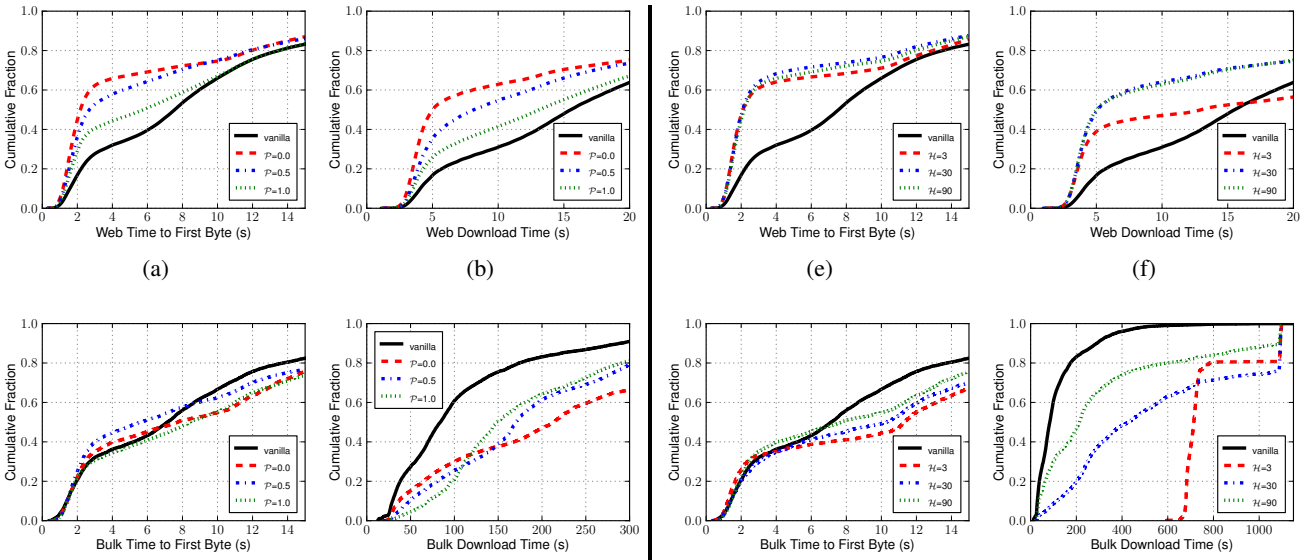
Figure 2 shows that it is possible to improve performance for interactive applications by statically tuning the `PerConnBWRate` and `PerConnBWBurst` parameters. However, it is also possible to hurt performance if these values are set incorrectly. We now explore throttling client connections in a way that adaptively adjusts the throttling rate based on information local to each relay to prevent configurations that hurt performance and reduce maintenance of the throttling algorithm.

Bit-splitting. Recall that our bit-splitting algorithm splits a relay's bandwidth fairly among client connections. Figure 3 shows the results of our bit-splitting algorithm as compared to vanilla Tor with no throttling. We see slight improvements for web clients for both time to first byte and overall download times, while download times for bulk clients are slightly increased. The effect is pleasant: a simple algorithm that does not hurt bulk clients too much but still improves performance for everyone else. However, the bandwidth not used by quiet connections is essentially wasted. Although we did not observe this in our simulations, it is possible that web client performance could be harmed. For example, if a relay has a large enough number



(a) (b) (c) (d)

Figure 3: Throttling clients by splitting bits. Responsiveness (a) and throughput (c) both improve for web clients at the expense of bulk throughput (d).



(a) (b) (e) (f)

(c) (d) (g) (h)

Figure 4: Throttling clients by flagging bulk connections. Flagged connections are rate-limited at 5 KiBps until their cell count EWMA reaches fraction $\mathcal{P}$ of the fair bandwidth cell count EWMA. (a)–(d) use a constant per-connection half-life $\mathcal{H}$ of 90 seconds and varies $\mathcal{P}$, affecting the length of the throttling period, and (e)–(h) vary $\mathcal{H}$, affecting the ability to differentiate bulk from web connections, while keeping $\mathcal{P}$ constant at 0.0.

of connections and none of them contain bulk traffic, then the algorithm would waste bandwidth on the quiet connections and throttle the busy connections, but never throttle bulk traffic (see Section 5).

Flagging Unfair Clients. Recall that our flagging algorithm adaptively flags and throttles bulk connections based on a connection-level cell EWMA. The results from our flagging algorithm experiments are shown in Figure 4. All experiments are run with a flagged connection throttle rate of 5 KiBps. Figures 4a–4d show the effect that changing the parameter $\mathcal{P}$ has on client performance, while fixing the half-life $\mathcal{H}$ at 90 seconds. Note that $\mathcal{P} = 0.0$ indicates that when a connection is flagged, it remains flagged for the remainder of its existence. Further, $\mathcal{P} = 1.0$ will cause a flagged connection's EWMA to hover around the meta-EWMA, assuming it is constantly transferring data. As expected, we find that lower $\mathcal{P}$ values result in more aggressive throttling of bulk connections and therefore better responsiveness (Figure 4a) and throughput (Figure 4b) for web clients.

Figures 4e–4h show the effect that changing the parameter $\mathcal{H}$ has on client performance, while fixing fraction $\mathcal{P}$ at 0.0. As we increase $\mathcal{H}$, the EWMA alogrithm will "remember" bytes that were sent further into the past. Small $\mathcal{H}$, e.g. 3 seconds, means that the EWMA will be high even if the connection only started transferring data seconds ago. As our results show in Figure 4h, all bulk connections are quickly flagged

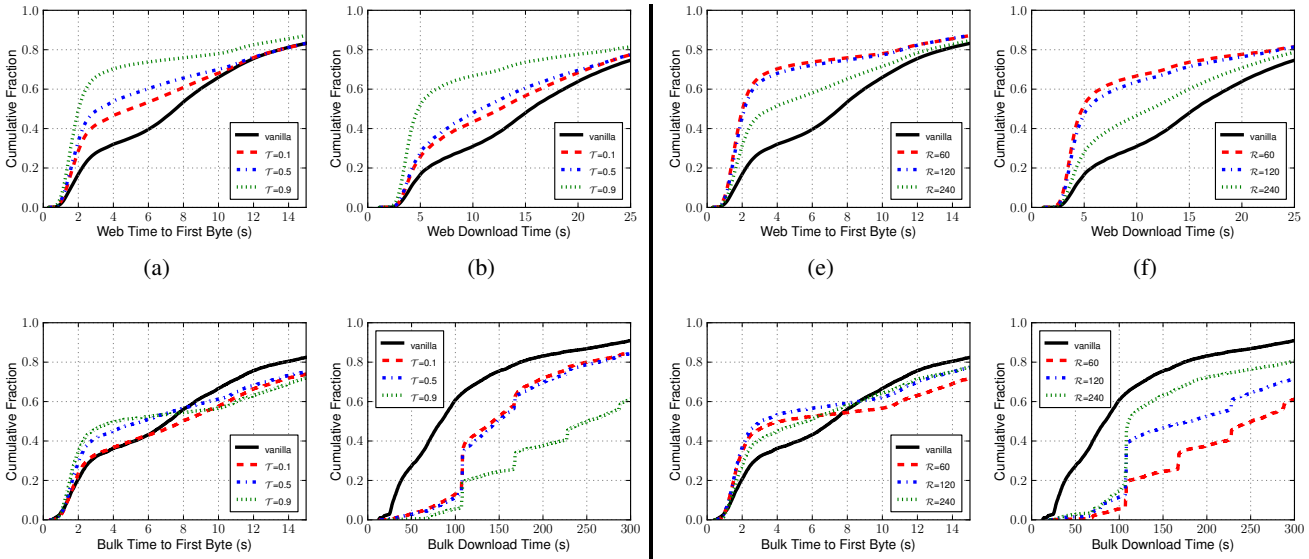


Figure 5: Throttling clients using thresholds. The threshold $\mathcal{T}$ configures the loudest fraction of connections that get throttled, while the throttle rate is obtained from the average throughput of the $\mathcal{T}^{th}$ loudest connection over the last $\mathcal{R}$ seconds. (a)–(d) show performance while varying $\mathcal{T}$ as shown with a constant $\mathcal{R}$ of 60 seconds. (e)–(h) hold $\mathcal{T}$ constant at 0.9 while varying $\mathcal{R}$ as shown. A 90 percent threshold and 60 second refresh period shows the best performance for interactive clients in our simulations while flooring the throttle rate at $\mathcal{F}$ = 50 KiBps.

when $\mathcal{H}$ = 3 seconds. Bulk downloads complete between roughly 600-1200 seconds, since bulk connections get their first 2 MiB of the download without being throttled. Unfortunately, Figure 4f shows that $\mathcal{H}$ = 3 also means we incorrectly flag some web connections as bulk because their EWMA over the short term is similar to that of a bulk connection.

Increasing $\mathcal{H}$ means creating a larger distinction between bulk and web clients and Figure 4h shows two interesting effects. First, larger $\mathcal{H}$ means it takes longer for a connection to become flagged, so download times drastically improve over all bulk clients. Second, larger $\mathcal{H}$ means that some bulk connections may never get flagged. This would happen if the entry node is receiving data from an upstream bottleneck, the rate from which is low enough to fall under the entry node's meta-EWMA of the client connection. So even though the exit or middle relay would have been able to flag the connection, this information is lost since our algorithm only throttles client connections at the entry node. The difference in web client performance between $\mathcal{H}$ = 30 seconds and $\mathcal{H}$ = 90 seconds is small in Figure 4f since in both cases, entry node bottlenecks are reduced and web connections are not being throttled. We suggest that 90 seconds is an appropriate setting for $\mathcal{H}$ since this will still throttle bulk connections at the worst of bottlenecks and is least likely to throttle web connections. Further, Figure 4h shows that $\mathcal{H}$ = 90 seconds is less intrusive on bulk connections.

The experimental results of our flagging algorithm show that the most aggressive throttling reduces download time for web clients by half over vanilla Tor in the median case. Further, Tor relay operators may adjust $\mathcal{P}$ to affect the level of throttling as they see fit. As long as $\mathcal{H}$ is set large enough, it would be difficult to misconfigure the algorithm in a way that harms web clients.

Throttling Using Thresholds. Recall that our threshold algorithm selects the loudest threshold of connections, and throttles them at the rate of the connection given by the threshold. Figure 5 shows the performance of the threshold throttling algorithm under various configurations. Our experiments use a constant throttle rate floor $\mathcal{F}$ so that no connection is ever throttled below $\mathcal{F} = 50$ KiBps. The experiments in Figures 5a–5d show performance under various thresholds $\mathcal{T}$ while holding the refresh period $\mathcal{R}$ constant at 60 seconds.

Figure 5d shows that bulk connections are throttled more aggressively as the threshold $\mathcal{T}$ increases from

0.5 to 0.9. As a result, both responsiveness in Figure 5a and throughput in Figure 5b drastically improve for interactive web clients. The reason is that $\mathcal{T} = 0.9$ ensures that all bulk connections in our simulations are included in the set of connections that get throttled. Although some web connections may also be included in this set, the adaptively selected throttle rate will be 50 KiBps or greater – more than enough to ensure web connections' token buckets are never completely depleted. However, if the threshold is set too low, then performance for interactive clients is not significantly affected. More importantly, it is not possible to *decrease* performance for interactive clients if the threshold is misconfigured.

The experiments in Figures 5e–5h show performance under a constant threshold $\mathcal{T} = 0.9$ while using various refresh periods $\mathcal{R}$. Figure 5h shows that as the period between throttle rate adjustments increases, the amount of bulk connection throttling decreases. This effect occurs because relays are less responsive to bulk traffic flows, i.e. bulk connections may send data for a longer period of time before being throttled. As shown in Figures 5e and 5f, the longest refresh period results in a more congested network as a whole and less of a performance improvement for web clients. Although not shown, if $\mathcal{R}$ is configured to less than 60 seconds, the algorithm computes less accurate long-term average thoughput for connections since there is less information for the calculation, affecting the adaptive throttle rate chosen by relays and the degree of performance improvements. We found $\mathcal{R} = 60$ seconds to be an appropriate setting in our simulations.

Algorithm Comparison. We have tested a wide range of parameters for various client throttling algorithms. Each algorithm offers unique and novel features, which Figure 6 directly compares. For each algorithm, we select the parameters we tested earlier in this section that maximize web client performance. Our comparison is done under different network loads: the number of bulk clients are adjusted to 25 in Figures 6a–6c, 50 in Figures 6d–6f (the same configuration as the experiments earlier in this section), and 100 in Figures 6g–6i.

We find that each algorithm is effective at throttling bulk clients independent of network load, shown in Figures 6c, 6f and 6i. However, performance benefits for web clients varies slightly as network load changes. When the number of bulk clients is halved, throughput in Figure 6b is fairly similar across algorithms. However, when the number of bulk clients is doubled, responsiveness in Figure 6g and throughput in Figure 6h 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 while bit-splitting adjusts the throttle rate less effectively than our threshold algorithm.

We see in Figures 6a, 6d, and 6g that 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 6c, 6f, and 6i show that they are also most aggressive at throttling bulk clients. The flagging algorithm appears very effective at accurately flagging bulk connections regardless of network 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 maintenence while providing significant performance improvements for web clients.

While requiring little maintenence, our algorithms were designed to use only information local to a single relay. Therefore, they are incrementally deployable and each relay operator may choose the desired throttling algorithm independent of other network relays. Our algorithms are already implemented in Tor and software patches are being prepared so they might be included in future Tor releases.

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 attack. We are interested in what an adversary can learn when guards throttle clients and how that information leakage affects the anonymity of the system. Before exploring practical attacks, we introduce two techniques an adversary may use to gather information about the network

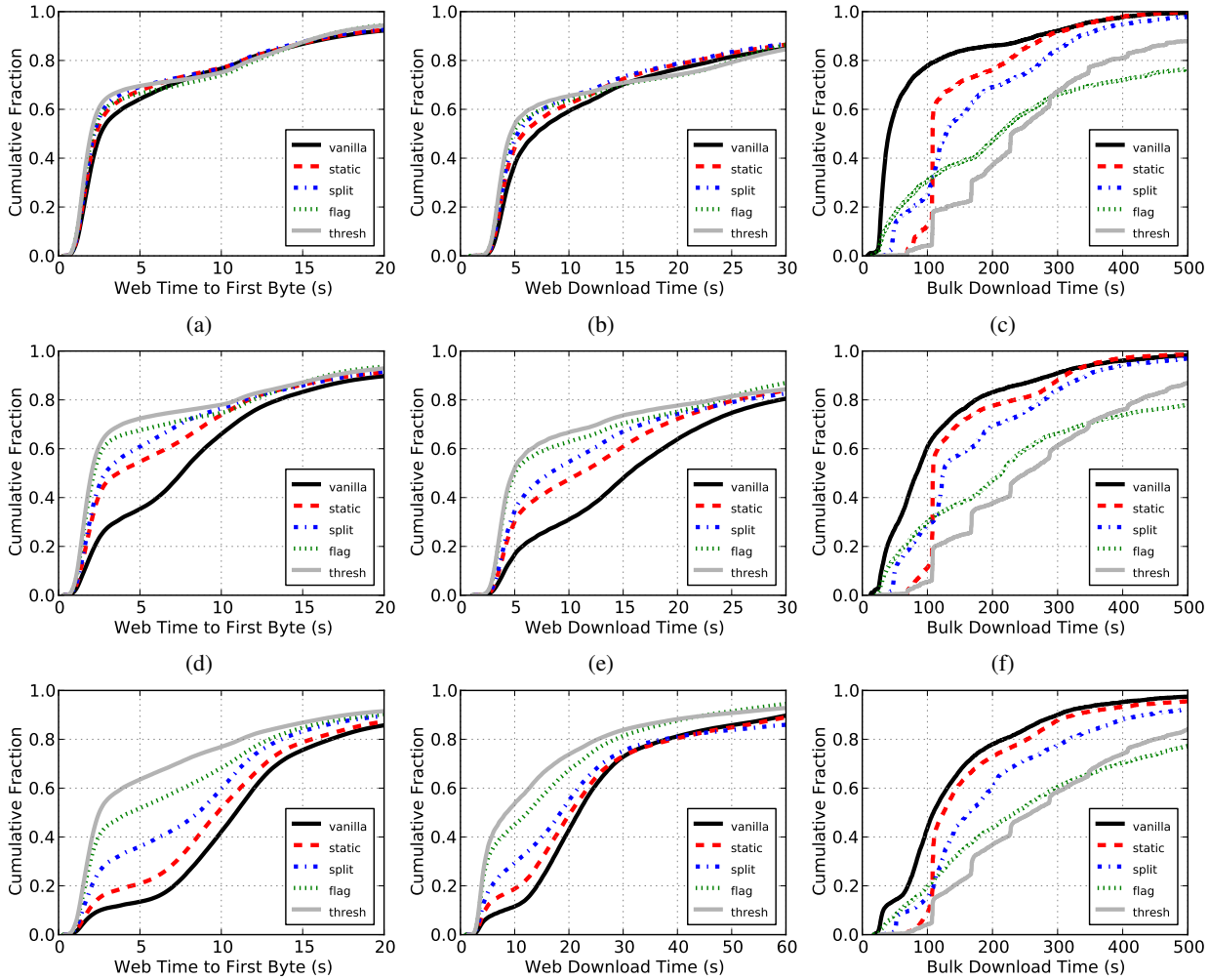


Figure 6: Comparison of client performance for each throttling algorithm and vanilla Tor, under various load. All experiments use 950 web clients while bulk clients are adjusted to 25 for (a)–(c), left at the default of 50 for (d)–(f), and adjusted to 100 for (g)–(i). The parameters for each algorithm are tuned to maximize client performance, according to experiments presented in Section 4.

given that a generic throttling algorithm is enabled at all guard nodes. Discussion about the security of specific throttling algorithms in the context of practical attacks will follow.

5.1 Gathering Information

Probing Guards. 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 bytes of the download.

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 λ on the client's connection to the guard can be approximated as the mean throughput over the remaining downloading interval:

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

We simulate probing in Shadow [26] to show its effectiveness against the static throttling algorithm. As apparent in Figure 7, the throttle rate was configured at 5 KiBps and the burst at 2 MiB. With enough resources, an adversary may probe every guard node to form a complete distribution 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.* [13]. 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.

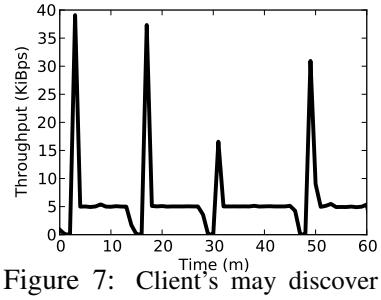


Figure 7: Client's may discover the throttle rate by probing guards.

5.2 Adversarial Attacks

We now explore several adversarial attacks in the context of client throttling algorithms.

Attack 1. In our first attack, an adversary obtains a distribution of throttle rates by probing all guards. We assume the adversary has resources to perform such an attack, e.g. by utilizing a botnet or other distributed network like PlanetLab [4]. 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 that of the target circuit or higher. By running the attack repeatedly over time, an intersection will narrow the set to those consistent with the target circuit at all measured times.

The flagging algorithm throttles all flagged connections to the same rate systemwide. (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 that rate is either flagged at some guard or just coincidentally at that rate and not flagged because its EWMA has remained below the `splitRate` 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 threshold algorithm combines the potential for information going to the attacker of each of the above two. 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 narrow further. 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. Our next attack improves the situation for the adversary considerably.

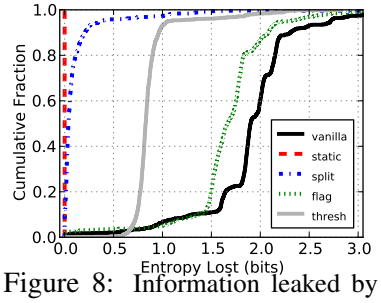


Figure 8: Information leaked by learning circuit throughputs.

We simulated a *crude* approximation of this attack in Shadow [26]. Our simulation allows us to form a distribution of the throttle rate at which a connection would become throttled had it acheived that rate. This approximates an adversary probing all guards. We then form a distribution of 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 is shown in Figure 8. We can see that the static algoirthm actually loses no information, since all connections are throttled to the same rate, while vanilla Tor without throttling actually loses *more* information based only on the bandwidth of each guard node.

Attack 2. As in Attack 1, the adversary again obtains a distribution of 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 of 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 dataset as Attack 1. Figure 9 shows that a connection's throttle rate leaks more information than its throughput. As in Attack 1, the guards bandwidth in our simulation leaks more information than the throttle rate of each connection.

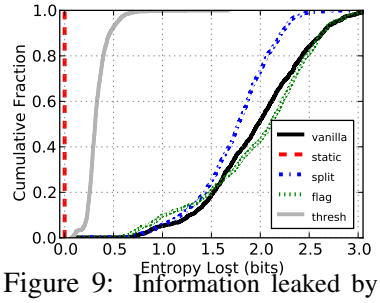


Figure 9: Information leaked by learning guards' throttle rates.

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, and can be used to confirm a circuit's guard node with high probability. An adversary in control of a malicious server may monitor the throughput of a target circuit over time. Then, the adversary may 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. 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.

6 Related Work

Relay Selection. Recent work on improving Tor's performance covers a wide range of topics. Snader and Borisov [27] suggest an algorithm where relays opportunistically measure their peers' performance, allowing clients to use empirical aggregations to select relays for their circuits rather than relying on self-reported bandwidth capabilities. Built into the algorithm is a user-tunable mechanism for selecting relays: clients may adjust how often the fast relays get chosen, trading off anonymity and performance while not significantly reducing either. This approach increases accuracy of available bandwidth estimates and reaction time to changes in network load while decreasing vulnerabilities to low-resource routing attacks.

Transport. Tor's performance has also been analyzed at the socket level, with suggestions for turning to UDP-based mechanisms for data delivery [33] or using a user-level TCP stack over a DTLS tunnel [22]. While the current Tor implementation multiplexes all circuits into a single kernel-level TCP stream to control information leakage, the TCP-over-DTLS approach suggests separate user-level TCP streams for each circuit and sends all TCP streams between two relays over a single kernel-level DTLS-secured [18] UDP socket. As a result, a circuit's TCP window is not unfairly reduced when other high-bandwidth circuits cause queuing delays and dropped packets.

Scheduling. Alternative scheduling approaches have recently gained interest. Tang and Goldberg [28] 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. In BRAIDS [14], Jansen *et al.* investigate new schedulers based on the proportional differentiation model [6] 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. Simulations show a decrease in latency for web clients and an increase in throughput for bulk clients, but requires a mechanism (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 [32]. Tschorsch and Scheuermann suggest that a round-robin scheduler could approximate a max-min algorithm [9] by choosing among all circuits rather than all TCP connections. More work is required to determine the suitability of this approach to a network-wide deployment.

Congestion. Improving performance and reducing congestion has been studied by taking an in-depth look at Tor's circuit and stream windows [1]. 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.

7 Conclusion

We discuss client throttling by guard relays as a way of reducing congestion throughout the Tor network. We explore static throttling 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 using local information. Our adaptive algorithms are considerably more effective than static throttling, especially when network load changes. We find that client throttling is effective at reducing congestion

and improving performance for web clients. We also analyzed the effects throttling has on anonymity and discussed the security of our algorithms against adversarial attack. We find that a guard's bandwidth leaks more information about its circuits when *not* throttling than any employed throttling algorithm. In addition to modifications to our current algorithms that optimize performance, we intend to further explore and analyze the impacts throttling has on anonymity in Tor.

Acknowledgments. We thank Roger Dingledine for helpful discussions regarding this work. This research was supported by NFS grant CNS-0917154, ONR, and DARPA.

References

[1] 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 International Symposium on Privacy Enhancing Technologies (PETS'11)*, 2011.

[2] A. Back, U. Moller, and A. Stiglic. Traffic analysis attacks and trade-offs in anonymity providing systems. In *Proceedings of Information Hiding Workshop (IH'01)*, pages 245–257, 2001.

[3] F. Chen and M. Perry. Improving Tor Path Selection. https://gitweb.torproject.org/torspec.git/blob/HEAD:/proposals/151-path-selection-improvements.txt.

[4] B. Chun, D. Culler, T. Roscoe, A. Bavier, L. Peterson, M. Wawrzoniak, and M. Bowman. Planetlab: an overlay testbed for broad-coverage services. *SIGCOMM Computer Communication Review*, 33:3–12, 2003.

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

[6] C. Dovrolis, D. Stiliadis, and P. Ramanathan. Proportional differentiated services: delay differentiation and packet scheduling. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM'99)*, pages 109–120, 1999.

[7] N. Evans, R. Dingledine, and C. Grothoff. A practical congestion attack on Tor using long paths. In *Proceedings of the 18th USENIX Security Symposium*, pages 33–50, 2009.

[8] D. M. Goldschlag, M. G. Reed, and P. F. Syverson. Hiding Routing Information. In *Proceedings of Information Hiding Workshop (IH'96)*, pages 137–150, 1996.

[9] E. Hahne. Round-robin scheduling for max-min fairness in data networks. *IEEE Journal on Selected Areas in Communications*, 9(7):1024–1039, 1991.

[10] G. Hardin. The tragedy of the commons. *Science*, 162(3859):1243–1248, December 1968.

[11] F. Hernandez-Campos, K. Jeffay, and F. Smith. Tracking the evolution of web traffic: 1995-2003. In *The 11th IEEE/ACM International Symposium on Modeling, Analysis, and Simulation of Computer Telecommunications Systems (MASCOTS'03)*, pages 16–25, 2003.

[12] A. Hintz. Fingerprinting websites using traffic analysis. In *Proceedings of Privacy Enhancing Technologies Workshop (PET'02)*, pages 171–178, 2002.

[13] N. Hopper, E. Vasserman, and E. Chan-Tin. How much anonymity does network latency leak? *ACM Transactions on Information and System Security (TISSEC'10)*, 13(2):1–28, 2010.

[14] R. Jansen, N. Hopper, and Y. Kim. Recruiting new Tor relays with BRAIDS. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS'10)*, pages 319–328, 2010.

[15] The Libevent event notification library, version 2.0. http://monkey.org/~provos/libevent/.

[16] K. Loesing. Measuring the Tor network: Evaluation of client requests to directories. Technical report, Tor Project, 2009.

[17] D. McCoy, K. Bauer, D. Grunwald, T. Kohno, and D. Sicker. Shining light in dark places: Understanding the Tor network. In *Proceedings of the 8th International Symposium on Privacy Enhancing Technologies (PETS'08)*, pages 63–76, 2008.

[18] N. Modadugu and E. Rescorla. The design and implementation of datagram TLS. In *Proceedings of the 11th Network and Distributed System Security Symposium (NDSS'11)*, 2004.

[19] S. Murdoch and G. Danezis. Low-cost traffic analysis of Tor. In *IEEE Symposium on Security and Privacy*, pages 183–195, 2005.

[20] T.-W. J. Ngan, R. Dingledine, and D. S. Wallach. Building incentives into Tor. In *The Proceedings of Financial Cryptography (FC'10)*, 2010.

[21] J. Raymond. Traffic analysis: Protocols, attacks, design issues, and open problems. In *Designing Privacy Enhancing Technologies*, pages 10–29, 2001.

[22] J. Reardon and I. Goldberg. Improving Tor using a TCP-over-DTLS tunnel. In *Proceedings of the 18th USENIX Security Symposium*, 2009.

[23] M. Reed, P. Syverson, and D. Goldschlag. Anonymous connections and onion routing. *IEEE Journal on Selected Areas in Communications*, 16(4):482–494, 1998.

[24] A. Serjantov and P. Sewell. Passive attack analysis for connection-based anonymity systems. *Computer Security–ESORICS*, pages 116–131, 2003.

[25] Shadow Development Repositories. http://github.com/shadow/.

[26] Shadow Resources. http://shadow.cs.umn.edu/.

[27] R. Snader and N. Borisov. A tune-up for Tor: Improving security and performance in the Tor network. In *Proceedings of the 16th Network and Distributed Security Symposium (NDSS'08)*, 2008.

[28] C. Tang and I. Goldberg. An improved algorithm for Tor circuit scheduling. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, pages 329–339, 2010.

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

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

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

[32] F. Tschorsch and B. Scheuermann. Tor is unfair–and what to do about it. 2011.

[33] C. Viecco. UDP-OR: A fair onion transport design. In *Proceedings of Hot Topics in Privacy Enhancing Technologies (HOTPETS'08)*, 2008.

Appendices

A Throttling Under Various Load

In Section 4 we evaluated both static throttling and our three adaptive algorithms, and compared them under various load. Figure 10 shows the time to first byte results for bulk clients from the same experiments as Figure 6. Time to first byte shows the responsiveness of the network, and is less important for bulk clients than throughput.

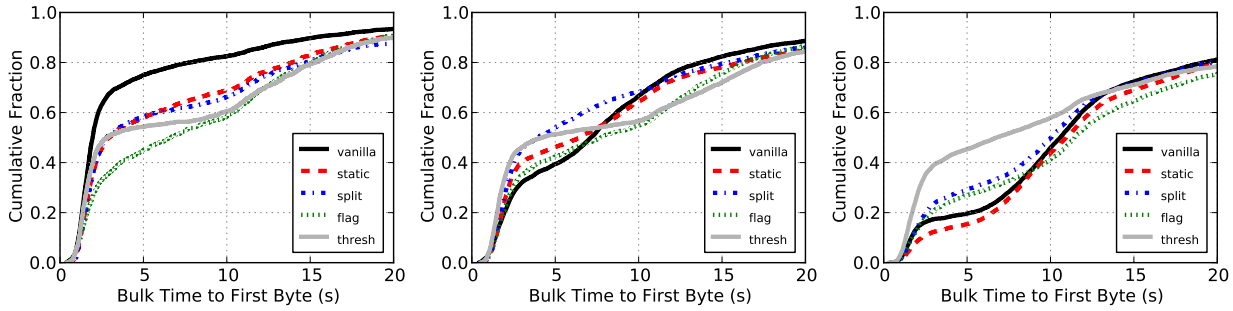


Figure 10: Comparison of responsiveness for bulk clients for each throttling algorithm and vanilla Tor, under various load. All experiments use 950 web clients while bulk clients are adjusted to 25 for (a), 50 for (b), and 100 for (c). The parameters for each algorithm are tuned to maximize client performance, according to experiments presented in Section 4.