Never Been KIST: Tor's Congestion Management Blossoms with Kernel-Informed Socket Transport

Rob Jansen[†] *John Geddes*[‡] *Chris Wacek*[*] *Micah Sherr*[*] *Paul Syverson*[†]

[†] *U.S. Naval Research Laboratory*
{*rob.g.jansen, paul.syverson*}*@nrl.navy.mil*

[‡] *University of Minnesota*
geddes@cs.umn.edu

[*] *Georgetown University*
{*cwacek, msherr*}*@cs.georgetown.edu*

Abstract

Tor's growing popularity and user diversity has resulted in network performance problems that are not well understood. A large body of work has attempted to solve these problems without a complete understanding of where congestion occurs in Tor. In this paper, we first study congestion in Tor at individual relays as well as along the entire end-to-end Tor path and find that congestion occurs almost exclusively in egress kernel socket buffers. We then analyze Tor's socket interactions and discover two major issues affecting congestion: Tor writes sockets sequentially, and Tor writes as much as possible to each socket. We thus design, implement, and test KIST: a new socket management algorithm that uses real-time kernel information to *dynamically compute the amount to write* to each socket while considering *all writable circuits* when scheduling new cells. We find that, in the medians, KIST reduces circuit congestion by over 30 percent, reduces network latency by 18 percent, and increases network throughput by nearly 10 percent. We analyze the security of KIST and find an acceptable performance and security trade-off, as it does not significantly affect the outcome of well-known latency and throughput attacks. While our focus is Tor, our techniques and observations should help analyze and improve overlay and application performance, both for security applications and in general.

1 Introduction

Tor [21] is the most popular overlay network for communicating anonymously online. Tor serves millions of users daily by transferring their traffic through a source-routed *circuit* of three volunteer relays, and encrypts the traffic in such a way that no one relay learns both its source and intended destination. Tor is also used to resist online censorship, and its support for hidden services, network bridges, and protocol obfuscation has helped attract a large and diverse set of users.

While Tor's growing popularity, variety of use cases, and diversity of users have provided a larger anonymity set, they have also led to performance issues [23]. For example, it has been shown that roughly half of Tor's traffic can be attributed to BitTorrent [18, 43], while the more recent use of Tor by a botnet [29] has further increased concern about Tor's ability to utilize volunteer resources to handle a growing user base [20, 36, 37, 45].

Numerous proposals have been made to battle Tor's performance problems, some of which modify the mechanisms used for path selection [13, 59, 60], client throttling [14, 38, 45], circuit scheduling [57], and flow/congestion control [15]. While some of this work has or will be incorporated into the Tor software, none of it has provided a comprehensive understanding of *where* the most significant source of congestion occurs in a complete Tor deployment. This lack of understanding has led to the design of uninformed algorithms and speculative solutions. In this paper, we seek a more thorough understanding of congestion in Tor and its effect on Tor's security. We explore an answer to the fundamental question—"*Where* is Tor slow?"—and design informed solutions that not only decrease congestion, but also improve Tor's ability to manage it as Tor continues to grow.

Congestion in Tor: We use a multifaceted approach to exploring congestion. First, we develop a shared library and Tor software patch for measuring congestion *local to relays* running in the public Tor network, and use them to measure congestion from three live relays under our control. Second, we develop software patches for Tor and the open-source Shadow simulator [7], and use them to measure congestion along the *full end-to-end path* in the largest known, at-scale, private Shadow-Tor deployment. Our Shadow patches ensure that our congestion measurements are accurate and realistic; we show how they significantly improve Shadow's TCP implementation, network topology, and Tor models.[1]

[1] We have contributed our patches to the Shadow project [7] and they have been integrated as of Shadow release `1.9.0`.

To the best of our knowledge, we are the first to consider such a comprehensive range of congestion information that spans from individual application instances to full network sessions for the entire distributed system. Our analysis indicates that congestion occurs almost exclusively inside of the kernel egress socket buffers, dwarfing the Tor and the kernel ingress buffer times. This finding is consistent among all three public Tor relays we measured, and among relays in every circuit position in our private Shadow-Tor deployment. This result is significant, as Tor does not currently prevent, detect, or otherwise manage kernel congestion.

Mismanaged Socket Output: Using this new understanding of *where* congestion occurs, we analyze Tor's socket output mechanisms and find two significant and fundamental design issues: Tor *sequentially* writes to sockets while ignoring the state of all sockets other than the one that is currently being written; and Tor writes *as much as possible* to each socket.

By writing to sockets sequentially, Tor's circuit scheduler considers only a small subset of the circuits with writable data. We show how this leads to improper utilization of circuit priority mechanisms, which causes Tor to send lower priority data from one socket *before* higher priority data from another. This finding confirms evidence from previous work indicating the ineffectiveness of circuit priority algorithms [35].

By writing as much as possible to each socket, Tor is often delivering to the kernel more data than it is capable of sending due to either physical bandwidth limitations or throttling by the TCP congestion control protocol. Not only does writing too much increase data queuing delays in the kernel, it also further reduces the effectiveness of Tor's circuit priority mechanisms because Tor relinquishes control over the priority of data after it is delivered to the kernel.[2] This kernel overload is exacerbated by the fact that a Tor relay may have thousands of sockets open at any time in order to facilitate data transfer between other relays, a problem that may significantly worsen if Tor adopts recent proposals [16, 26] that suggest increasing the number of sockets between relays.

KIST: Kernel-Informed Socket Transport: To solve the socket management problems outlined above, we design KIST: a Kernel-Informed Socket Transport algorithm. KIST has two features that work together to significantly improve Tor's control over network congestion. First, KIST changes Tor's circuit level scheduler so that it chooses from *all* circuits with writable data rather than just those belonging to a single TCP socket. Second, to complement this global scheduling approach, KIST also dynamically manages the amount of data written to each socket based on real-time kernel and TCP state information. In this way, KIST attempts to minimize the amount of data that exists in the kernel that cannot be sent, and to maximize the amount of time that Tor has control over data priority.

We perform in-depth experiments in our at-scale private Shadow-Tor network, and we show how KIST can be used to relocate congestion from the kernel into Tor, where it can be properly managed. We also show how KIST allows Tor to correctly utilize its circuit priority scheduler, reducing download latency by over 660 milliseconds, or 23.5 percent, for interactive traffic streams typically generated by web browsing behaviors.

We analyze the security of KIST, showing how it affects well-known latency and throughput attacks. In particular, we show the extent to which the latency improvements reduce the number of round-trip time measurements needed to conduct a successful latency attack [31]. We also show how KIST does not significantly affect an adversary's ability to collect accurate measurements required for the throughput correlation attack [44] when compared to vanilla Tor.

Outline of Major Contributions: We outline our major contributions as follows:

- in Section 3 we discuss improvements to the open-source Shadow simulator that significantly enhance its accuracy, including experiments with the largest known private Tor network of 3,600 relays and 13,800 clients running real Tor software;
- in Section 4 we discuss a library we developed to measure congestion in Tor, and results from the first known end-to-end Tor circuit congestion analysis;
- in Section 5 we show how Tor's current management of sockets results in ineffective circuit priority, detail the KIST design and prototype, and show how it improves Tor's ability to manage congestion through a comprehensive and full-network evaluation; and
- in Section 6 we analyze Tor's security with KIST by showing how our performance improvements affect well-known latency and throughput attacks.

2 Background and Related Work

Tor [21] is a volunteer-operated anonymity service used by an estimated hundreds of thousands of daily users [28]. Tor assumes an adversary who can monitor a portion of the underlying Internet and/or operate Tor relays. People primarily use Tor to prevent an adversary from discovering the endpoints of their communications, or disrupting access to information.

Tor Traffic Handling: Tor provides anonymity by forming source-routed paths called *circuits* that consist of (usually) three relays on an overlay network. Clients transfer TCP-based application traffic within these circuits; encrypted application-layer headers and payloads

[2] To the best of our knowledge, the Linux kernel uses a variant of the first-come first-serve queuing discipline among sockets.

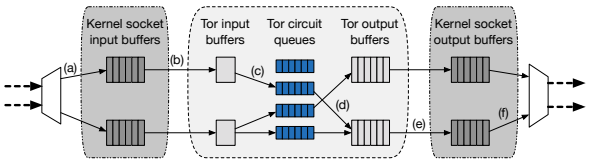


Figure 1: Internals of cell processing within in a Tor relay. Dashed lines denote TCP connections. Transitions between buffers—both within the kernel (side boxes) and within Tor (center box)—are shown with solid arrows.

make it more difficult for an adversary to discern an intercepted communication's endpoints or learn its plaintext.

A given circuit may carry several Tor *streams*, which are logical connections between clients and destinations. For example, a HTTP request to usenix.org may result in several Tor streams (i.e., to fetch embedded objects); these streams may all be transported over a single circuit. Circuits are themselves multiplexed over TLS connections between relays whenever their paths share an edge; that is, all concurrent circuits between relays u and v will be transferred over the same TLS connection between the two relays, irrespective of the circuits' endpoints.

The unit of transfer in Tor is a 512-byte *cell*. Figure 1 depicts the internals of cell processing within a Tor relay. In this example, the relay maintains two TLS connections with other relays. Incoming packets from the two TCP streams are first demultiplexed and placed into kernel socket input buffers by the underlying OS (Figure 1a)[3]. The OS processes the packets, usually in FIFO order, delivering them to Tor where they are reassembled into TLS-encrypted cells using dedicated Tor input buffers (Figure 1b). Upon receipt of an entire TLS datagram, the TLS layer is removed, the cell is onion-crypted,[4] and then transferred and enqueued in the appropriate Tor circuit queue (Figure 1c). Each relay maintains a queue for each circuit that it is currently serving. Cells from the same Tor input buffer may be enqueued in different circuit queues, since a single TCP connection between two relays may carry multiple circuits.

Tor uses a priority-based circuit scheduling approach that attempts to prioritize interactive web clients over bulk downloaders [57]. The circuit scheduler selects a cell from a circuit queue to process based on this prioritization, onion-crypts the cell, and stores it in a Tor output buffer (Figure 1d). Once the Tor output buffer contains sufficient data to form a TLS packet, the data is written to the kernel for transport (Figure 1f).

Improving Tor Performance: There is a large body of work that attempts to improve Tor's network performance, e.g., by refining Tor's relay selection strategy [11,55,56] or providing incentives to users to operate Tor relays [36, 37, 45]. These approaches are orthogonal and can be applied in concert with our work, which focuses on improving Tor's congestion management.

Most closely related to this paper are approaches that modify Tor's circuit scheduling, flow control, or transport mechanisms. Reardon and Goldberg suggest replacing Tor's TCP-based transport mechanism with UDP-based DTLS [54], while Mathewson explores using SCTP [40]. Murdoch [47] explains that the UDP approach is promising, but there are challenges that have thus far prevented the approach from being deployed: there is limited kernel support for SCTP; and the lack of hop-by-hop reliability from UDP-based transports causes increased load at Tor's exit relays. Our work allows Tor to best utilize the existing TCP transport in the short term while work toward a long term UDP deployment strategy continues.

Tang and Goldberg propose the use of the exponential weighted moving average (EWMA) to characterize circuits' recent levels of activity, with bursty circuits given greater priority than busy circuits [57] (to favor interactive web users over bulk downloaders). Unfortunately, although Tor has adopted EWMA, the network has not significantly benefitted from its use [35]. In our study of *where* Tor is slow, we show that EWMA is made ineffective by Tor's current management of sockets, and can be made effective through our proposed modifications.

AlSabah et al. propose an ATM-like congestion and flow control system for Tor called N23 [15]. Their approach causes pushback effects to previous nodes, reducing congestion in the entire circuit. Our KIST strategy is complementary to N23, focusing instead on local techniques to remove kernel-level congestion at Tor relays.

Torchestra [26] uses separate TCP connections to carry interactive and bulk traffic, isolating the effects of congestion between the two traffic classes. Conceptually, Torchestra moves circuit-selection logic to the kernel, where the OS schedules packets for the two connections. Relatedly, AlSabah and Goldberg introduce PCTCP [16], a transport mechanism for Tor in which each circuit is assigned its own IPsec tunnel. In this paper, we argue that overloading the kernel with additional sockets reduces the effectiveness of circuit priority mechanisms since the kernel has no information regarding the priority of data. In contrast, we aim to move congestion management to Tor, where priority scheduling can be most effective.

Nowlan et al. [50] propose the use of uTCP and uTLS [49] to tackle the "head-of-line" blocking problem in Tor. Here, they bypass TCP's in-order delivery mechanism to peek at traffic that has arrived but is not ready to be delivered by the TCP stack (e.g., because an earlier packet was dropped). Since Tor multiplexes multiple cir-

[3]For simplicity, we consider only relays that run Linux since such relays represent 75% of all Tor relays and contribute 91% of the bandwidth of the live Tor network [58].

[4]Encrypted or decrypted, depending on circuit direction.

cuits over a single TCP connection, their technique offers significant latency improvements when connections are lossy, since already-arrived traffic can be immediately processed. Our technique can be viewed as a form of application-layer head-of-line countermeasure since we move scheduling decisions from the TCP stack to within Tor. In contrast to Nowlan et al.'s approach, we do not require any kernel-level modifications or changes to Tor's transport mechanism.

3 Enhanced Network Experimentation

To increase confidence in our experiments, we introduce three significant enhancements to the Shadow Tor simulator [35] and its existing models [33]: a more realistic simulated kernel and TCP network stack, an updated Internet topology model, and the largest known deployed private Tor network. The enhancements in this section represent a large and determined engineering effort; we will show how Tor experimental accuracy has significantly benefited as a result of these improvements. We remark that our improvements to Shadow will have an immediate impact beyond this work to the various research groups around the world that use the simulator.

Shadow TCP Enhancements: After reviewing Shadow [7], we first discovered that it was missing many important TCP features, causing it to be less accurate than desired. We enhanced Shadow by adding the following: retransmission timers [52], fast retransmit/recovery [12], selective acknowledgments [42], and forward acknowledgments [41]. Second, we discovered that Shadow was using a very primitive version of the basic additive-increase multiplicative-decrease (AIMD) congestion control algorithm. We implemented a much more complete version of the CUBIC algorithm [27], the default congestion control algorithm used in the Linux kernel since version 2.6.19. CUBIC is an important algorithm for properly adjusting the congestion window. We will show how our implementation of these algorithms greatly enhance Shadow's accuracy, which is paramount to the remainder of this paper. See Appendix A.1 [34] for more details about our modifications.

We verify the accuracy of Shadow's new TCP implementation to ensure that it is adequately handling packet loss and properly growing the congestion window by comparing its behavior to ns [51], a popular network simulator, because of the ease at which ns is able to model packet loss rates. In our first experiment, both Shadow and ns have two nodes connected by a 10 MiB/s link with a 10 ms round trip time. One node then downloads a 100 MiB file 10 times for each tested packet loss rate. Figure 2a shows that the average download time in Shadow matches well with ns over varying packet loss rates. Although not presented here, we similarly validated Shadow with our changes against a real network link using the bandwidth and packet loss rate that was achieved over our switch; the results did not significantly deviate from those presented in Figure 2a.

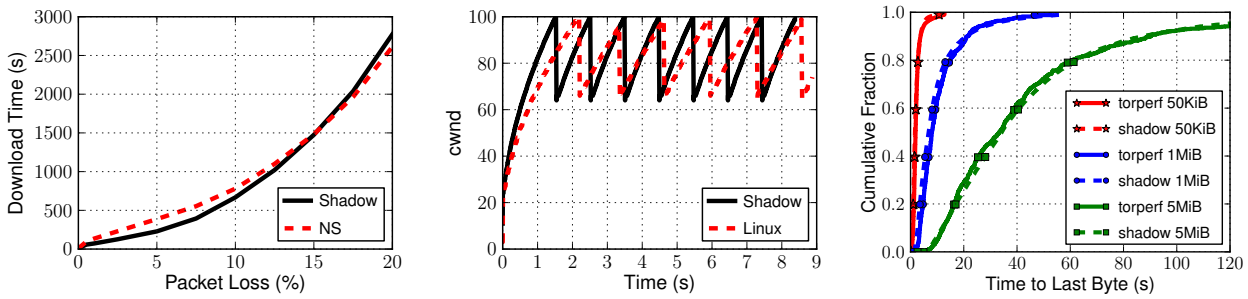
For our second experiment, we check that the growth of the congestion window using CUBIC is accurate. We first transfer a 100 MiB file over a 100 Mbit/s link between two physical Ubuntu 12.04 machines running the 3.2.0 Linux kernel. We record the `cwnd` (congestion window) and `ssthresh` (slow start threshold) values from the `getsockopt` function call using the `TCP_INFO` option. We then run an identical experiment in Shadow, setting the slow start threshold to what we observed from Linux and ensuring that packet loss happens at roughly the same rate. Figure 2b shows the value of `cwnd` in both Shadow and Linux over time, and we see almost identical growth patterns. The slight variation in the saw-tooth pattern is due to unpredictable variation in the physical link that was not reproduced by Shadow. As a result, Shadow's `cwnd` grew slightly faster than Linux's because Shadow was able to send one extra packet. We believe this is an artifact of our particular physical configuration and do not believe it significantly affects simulation accuracy in general: more importantly, the overall saw-tooth pattern matches well.

The two experiments discussed above give us high confidence that our TCP implementation is accurate, both in responding to packet loss and in operation of the CUBIC congestion control algorithm.

Shadow Topology Enhancements: To ensure that we are causing the most realistic performance and congestion effects possible during simulation, we enhance Shadow using techniques from recent research in modeling Tor topologies [39, 59], traceroute data from CAIDA [2], and client/server data from the Tor Metrics Portal [8] and Alexa [1]. This data-driven Internet map is more realistic than the one Shadow provides, and includes 699,029 vertices and 1,338,590 edges. For space reasons, we provide more details in Appendix A.2 [34].

Tor Model: Using Shadow with the improvements discussed above, we build a Tor model that reflects the real Tor network as it existed in July 2013, using the then-latest stable Tor version `0.2.3.25`. (We use this model for all experiments in this paper.) Using data from the Tor Metrics Portal [8], we configure a complete, private Tor network following Tor modeling best practices [33], and attach every node to the closest network location in our topology map. The resulting Tor network configuration includes 10 directory authorities, 3,600 relays, 13,800 clients, and 4,000 file servers—the largest known working private experimental Tor network, and the first to run at scale to the best of our knowledge.

The 13,800 clients in our model provide background traffic and load on the network. 10,800 of our clients



(a) Download Time, Shadow vs. ns (b) Congestion Window, Shadow vs. Linux (c) Download Time, Shadow-Tor vs. Tor

Figure 2: Figure 2a compares Shadow to ns download times. Figure 2b compares congestion window over time when Shadow and Linux have the same link properties. Figure 2c compares Shadow-Tor to live Tor measurements collected from Tor Metrics [8].

download 320 KiB files (the median size of a web page according to the most recent web statistics published by a Google engineer [53]) and then wait for a time chosen uniformly at random from the range [1, 60 000] milliseconds after each completed download. 1,200 of our clients repeatedly download 5 MiB files with no pauses between completing a download and starting the next. The ratio of these client behaviors was chosen according to the latest known measurements of client traffic on Tor [18, 43]. Shadow also contains 1,800 TorPerf [9] clients that download a file over a fresh circuit and pause for 60 seconds after each successful download. (TorPerf is a tool for measuring Tor performance.) 600 of the TorPerf clients download 50 KiB files, 600 download 1 MiB files, and 600 download 5 MiB files. Our simulations run for one virtual hour during each experiment.

Figure 2c shows a comparison of publicly available TorPerf measurements collected on the live Tor network [8] to those collected in our private Shadow-Tor network. As shown in Figure 2c, our full size Shadow-Tor network is extremely accurate in terms of time to complete downloads for all file sizes. These results give us confidence that our at-scale Shadow-Tor network is strongly representative of the deployed Tor network.

4 Congestion Analysis

In this section, we explore *where* congestion happens in Tor through a large scale congestion analysis. We take a multifaceted approach by measuring congestion as it occurs in both the live, public Tor network, and in an experimental, private Tor network running in Shadow. By analyzing relays in the public Tor network, we get the most realistic and accurate view of what is happening at our measured relays. We supplement the data from a relatively small public relay sample with measurements from a much larger set of private relays, collecting a larger and more complete view of Tor congestion.

To understand congestion, we are interested in measuring the time that data spends inside of Tor as well as inside of kernel sockets in both the incoming and outgoing directions. We will discuss our findings in both environments after describing the techniques that we used to collect the time spent in these locations.

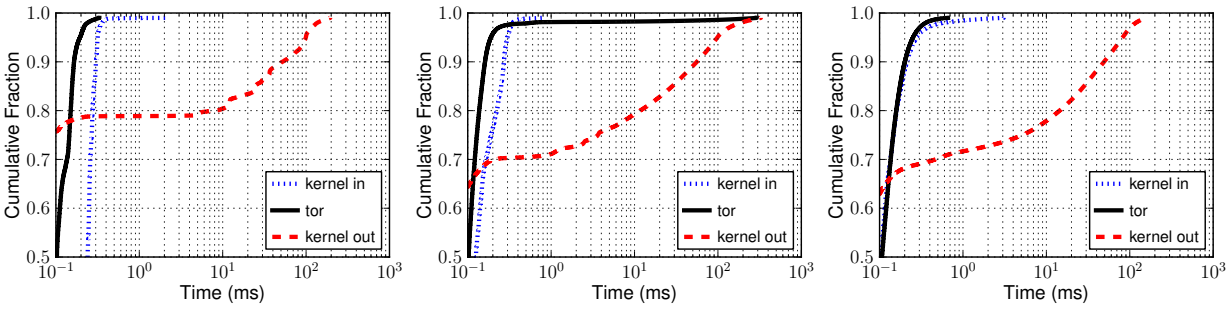
4.1 Congestion in the Live Tor Network

Relays running in the operational network provide the most accurate source of congestion data, as these relays are serving real clients and transferring real traffic. As mentioned above, we are interested in measuring queuing times inside of the Tor application as well as inside of the kernel, and so we developed techniques for both in the local context of a public Tor relay.

Tor Congestion: Measuring Tor queuing times requires some straightforward modifications to the Tor software. As soon as a relay reads the entire cell, it internally creates a cell structure that holds the cell's circuit ID, command, and payload. We add a new unique cell ID value. Whenever a cell enters Tor and the cell structure is created, we log a message containing the current time and the cell's unique ID. The cell is then switched to the outgoing circuit. After it's sent to the kernel we log another message containing the time and ID. The difference between these times represents Tor application congestion.

Kernel Congestion: Measuring kernel queuing times is much more complicated since Tor does not have direct access to the kernel internals. In order to log the times when a piece of data enters and leaves the kernel in both the incoming and outgoing directions, we developed a new, modular, application-agnostic, multithreaded library, called `libkqtime`.[5] `libkqtime` uses `libpcap` [6] to determine when data crosses the host/network boundary, and *function interposition* on

[5] `libkqtime` was written in 770 LOC, and is available for download as open source software [5].



(a) curiosity1 (b) curiosity2 (c) curiosity3

Figure 3: The distribution of congestion inside Tor and the kernel on our 3 relays running in the public Tor network, as measured between 2014-01-20 and 2014-01-28. Most congestion occurred in the outbound kernel queues on all three relays.

the `write()`, `send()`, `read()`, and `recv()` functions to determine when it crosses the application/kernel boundary. The library copies a 16 byte tag as data enters the kernel from either end, and then searches for the tag as data leaves the kernel on the opposite end. This process works in both directions, and the timestamps collected by the library allow us to measure both inbound and outbound kernel congestion. Appendix B [34] gives a more detailed description of `libkqtime`.

Results: To collect congestion information in Tor, we first ran three live relays (curiosity1, curiosity2, and curiosity3) using an unmodified copy of Tor release `0.2.3.25` for several months to allow them to stabilize. We configured them as non-exit nodes and used a network appliance to rate limit curiosity1 at 1 Mbit/s, curiosity2 at 10 Mbit/s, and curiosity3 at 50 Mbit/s. Only curiosity2 had the guard flag (could be chosen as entry relay for a circuit) during our data collection. On 2014-01-20, we swapped the Tor binary with a version linked to `libkqtime` and modified as discussed in Section 4.1. We collected Tor and kernel congestion for 190 hours (just under 8 days) ending on 2014-01-28, and then replaced the vanilla Tor binary.

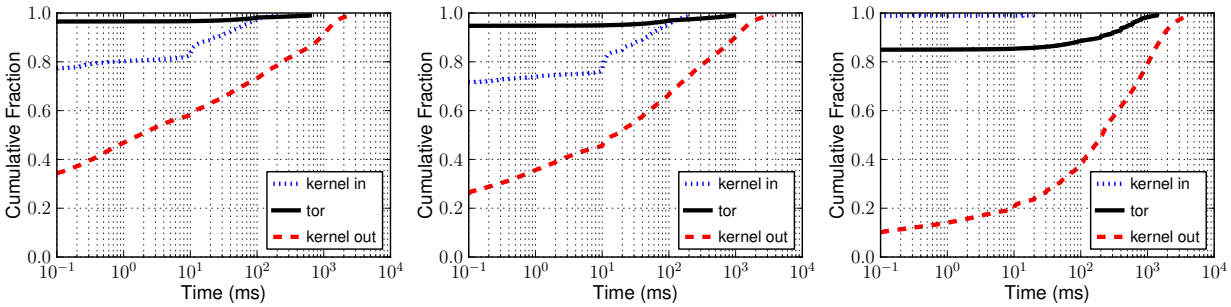
The distributions of congestion as measured on each relay during the collection period are shown in Figure 3 with logarithmic x-axes. Our measurements indicate that most congestion, when present, occurs in the *kernel outbound queues*, while kernel inbound and Tor congestion are both less than 1 millisecond for over 95 percent of our measurements. This finding is consistent across all three relays we measured. Kernel outbound congestion increases from curiosity1 to curiosity2, and again slightly from curiosity2 to curiosity3, indicating that congestion is a function of relay capacity or load. We leave it to future work to analyze the strength of this correlation, as that is outside the scope of this paper.

Ethical Considerations: We took careful protections to ensure that our live data collection did not breach users' anonymity. In particular, *we captured only buffered data timing information*; no network addresses were ever recorded. We discussed our experimental methodology with Tor Project maintainers, who raised no objections. Finally, we contacted the IRB of our relay host institution. The IRB decided that no review was warranted since our measurements did not, in their opinion, constitute human subjects research.

4.2 Congestion in a Shadow-Tor Network

While congestion data from real live relays is the most accurate, it only gives us a limited view of congestion local to our relays. The congestion measured at our relays may or may not be representative of congestion at other relays in the network. Therefore, we use our private Shadow-Tor network to supplement our congestion data and enhance our analysis. Using Shadow provides many advantages over live Tor: it's technically simpler; we are able to measure congestion *at all relays* in our private network; we can track the congestion of every cell *across the entire circuit* because we do not have privacy concerns with Shadow; and we can analyze how congestion changes with varying network configurations.

Tor and Kernel Congestion: The process for collecting congestion in Shadow is simpler than in live Tor, since we have direct access to Shadow's virtual kernel. In our modified Tor, each cell again contains a unique ID as in Section 4.1. However, when running in Shadow, we also add a 16 byte magic token and include both the unique ID and the magic token when sending cells out to the network. The unique ID is forwarded with the cell as it travels through the circuit. Since Shadow prevents Tor from encrypting cell contents for efficiency reasons, the Shadow kernel can search outgoing packets for the unencrypted magic token immediately before they leave the virtual network interface. When found, it logs the unique cell ID with a timestamp. It performs an analogous procedure for incoming packets immediately after they arrive on the virtual network interface. These



(a) Relays in Entry Position (b) Relays in Middle Position (c) Relays in Exit Position

Figure 4: Relay congestion by circuit position in our Shadow-Tor network, measured on on circuits using end-to-end cells from 1,200 clients selected uniformly at random. Most congestion occurred in the outbound kernel queues, independent of relay position.

Shadow timestamps are combined with the timestamps logged when a cell enters and leaves Tor to compute both Tor and kernel congestion.

Results: We use our model of Tor as described in Section 3, with the addition of the cell tracking information discussed above. Since tracking every cell would consume an extremely large amount of disk space, we sample congestion as follows: we select 10 percent of the non-TorPerf clients (1,200 total) in our network chosen uniformly, and track 1 of every 100 cells traveling over circuits they initiate. The tracking timestamps from these cells are then used to attribute congestion to the relays through which the cells are traveling.

It is important to understand that our method does not sample *relay* congestion uniformly: the congestion measurements will be biased towards relays that are chosen more often by clients, according to Tor's bandwidth-weighted path selection algorithm. This means that our results will represent the congestion that a typical *client* will experience when using Tor. We believe that these results are more meaningful than those we could obtain by uniformly sampling congestion at each relay independently (as we did in Section 4.1), because ultimately we are interested in improving clients' experience.

The distributions of congestion measured in Shadow for each circuit position are shown in Figure 4. We again find that congestion occurs most significantly in the kernel outbound queues, regardless of a relay's circuit position. Our Shadow experiments indicate higher congestion than in live Tor, which we attribute to our client-oriented sampling method described above.

5 Kernel-Informed Socket Transport

Our large scale congestion analysis from Section 4 revealed that the most significant delay in Tor occurs in outbound kernel queues. In this section, we first explore how this problem adversely affects Tor's traffic management by disrupting existing scheduling mechanisms to the extent that they become ineffective. We then describe the KIST algorithm and experimental results.

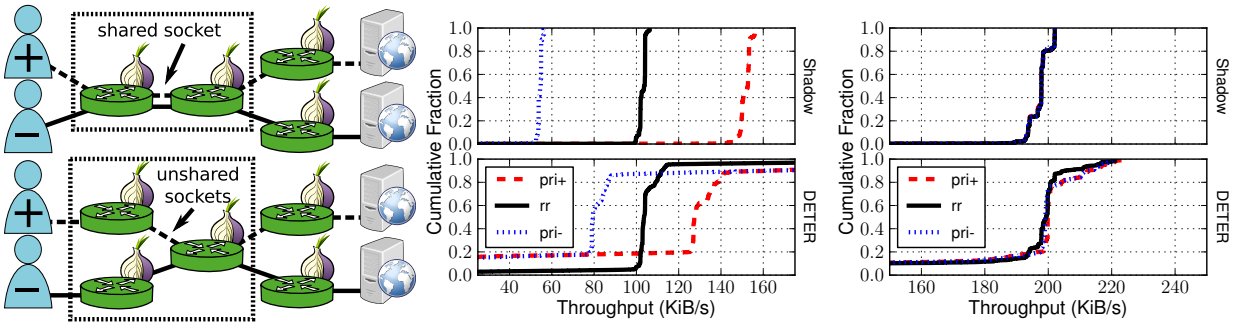
5.1 Mismanaged Socket Output

As described in Section 2, each Tor relay creates and maintains a single TCP connection to every relay to which it is connected. All communication between two relays occurs through this single TCP connection channel. In particular, this channel multiplexes all circuits that are established between its relay endpoints. TCP provides Tor a reliable and in-order data transport.

Sequential Socket Writes: Tor uses the asynchronous event library `libevent` [4] to assist with sending and receiving data to and from the kernel (i.e. network). Each TCP connection is represented as a socket in the kernel, and is identified by a unique socket descriptor. Tor registers each socket descriptor with libevent, which itself manages kernel polling and triggers an asynchronous notification to Tor via a callback function of the readability and writability of that socket. When Tor receives this notification, it chooses to read or write as appropriate.

An important aspect of these libevent notifications is that they happen for *one socket at a time*, regardless of the number of socket descriptors that Tor has registered. Tor attempts to send or receive data from that one socket without considering the state of any of the other sockets. This is particularly troublesome when writing, as Tor will only be able to choose from the non-empty circuits belonging to the currently triggered socket and no other. Therefore, Tor's circuit scheduler may schedule a circuit with worse priority than it would have if it could choose from all sockets that are able to be triggered at that time. Since the kernel schedules with a first-come first-serve (FCFS) discipline, Tor may actually be sending data out of priority order simply due to the order in which the socket notifications are delivered by libevent.

Bloated Socket Buffers: Linux uses TCP autotuning to dynamically and monotonically increase each



(a) Socket Sharing Scenarios (b) Throughput with Shared Socket (c) Throughput with Unshared Sockets

Figure 5: Socket sharing affects circuit priority scheduling.

socket buffer's capacity using the socket connection's bandwidth-delay product calculation [61]. TCP auto-tuning increases the amount of data the kernel will accept from the application in order to ensure that the socket is able to fully utilize the network link. TCP auto-tuning is an extremely useful technique to maximize throughput for applications with few sockets or without priority requirements. However, it may cause problems for more complex applications like Tor.

When libevent notifies Tor that a socket is writable, Tor writes as much data as possible to that socket (i.e., until the kernel returns `EWOULDBLOCK`). Although this improves utilization when only a few auto-tuned sockets are in use, consider a Tor relay that writes to thousands of auto-tuned sockets (a common situation since Tor maintains a socket for every relay with which it communicates). These sockets will *each* attempt to accept enough data to fully utilize the link. If Tor fills all of these sockets to capacity, the kernel will clearly be unable to immediately send it all to the network. Therefore, with many active sockets in general and for asymmetric connections in particular, the potential for kernel queuing delays are dramatic. As we have shown in Section 4, writing as much as possible to the kernel as Tor currently does results in large kernel queuing delays.

Tor can no longer adjust data priority once it is sent to the kernel, even if that data is still queued in the kernel when Tor receives data of higher importance later. To demonstrate how this may result in poor scheduling decisions, consider a relay with two circuits: one contains sustained, high throughput traffic of worse priority (typical of many bulk data transfers), while the other contains bursty, low throughput traffic of better priority (typical of many interactive data transfer sessions). In the absence of data on the low throughput circuit, the high throughput circuit will fill the entire kernel socket buffer whether or not the kernel is able to immediately send that data. Then, when a better priority cell arrives, Tor will immediately schedule and write it to the kernel. However, since the kernel sends data to the network in the same order in which it was received from the application (FCFS), that better priority cell data must wait until all of the previously received high throughput data is flushed to the network. This problem theoretically worsens as the number of sockets increase, suggesting that recent research proposing that Tor use multiple sockets between each pair of relays [16, 26] may be misguided.

Effects on Circuit Priority: To study the effects on circuit priority, we customized Tor as follows. First, we added the ability for each client to send a special cell after building a circuit that communicates one of two priority classes to the circuit's relays: a better priority class; or a worse priority class. Second, we customized the built-in EWMA circuit scheduler that prioritizes bursty traffic over bulk traffic [57] to include a priority factor $\mathscr{F}$: the circuit scheduler counts $\mathscr{F}$ cells for every cell scheduled on a worse priority circuit. Therefore, the EWMA of the worse priority class will effectively increase $\mathscr{F}$ times faster than normal, giving a scheduling advantage to better priority traffic.

We experiment with two separate private Tor networks: one using Shadow [35], a discrete event network simulator that runs Tor in virtual processes; and the other using DETER [3], a custom experimentation testbed that runs Tor on bare-metal hardware. We consider two clients downloading from two file servers through Tor in the scenarios shown in Figure 5a:

- *shared socket:* the clients share entry and middle relays, but use different exit relays – the clients' circuits each belong to the same socket connecting the middle to the entry; and
- *unshared sockets:* the clients share only the middle relay – the clients' circuits each belong to independent sockets connecting the middle to each entry.

We assigned one client's traffic to the better priority class (denoted with "+") and the other client's traffic to the worse priority class (denoted with "-"). We configured all nodes with a 10 Mbit symmetric access link, and approximated a middle relay bottleneck by setting its socket buffer size to 32 KiB. Our configuration allows us

to focus on the socket contention that will occur at the middle relay, and the four cases that result when considering whether or not the two circuits share incoming or outgoing TCP connections at the middle relay. Clients downloaded data through the circuits continuously for 10 minutes in Shadow and 60 minutes on DETER.[6]

The results collected during each of the scenarios are shown in Figure 5. Plotted is the cumulative distribution of the throughput achieved by the better ("pri+") and worse ("pri-") priority clients using the priority scheduler, as well as the combined cumulative distribution for both clients using Tor's default round-robin scheduler ("rr"). As shown in Figure 5b, performance differentiation occurs correctly with the priority scheduler on a shared socket. However, as shown in Figure 5c, the priority scheduler is unable to differentiate throughput when the circuits do not share a socket.

Discussion: As outlined above, the reason for no differentiation in the case of the unshared socket is that both circuits are treated independently by the scheduler due to the sequential libevent notifications and the fact that Tor currently schedules circuits belonging to one socket at a time while ignoring the others. We used TorPS [10], a Tor path selection simulator, to determine how often we would expect unshared sockets to occur in practice. We used TorPS to build 10 million paths following Tor's path selection algorithm, and computed the probability of two circuit paths belonging to each scenario. We found that any two paths may be classified as unshared (they share at least one relay but never share an outgoing socket) at least 99.775 percent of the time, clearly indicating that adjusting Tor's socket management may have a dramatic effect on data priority inside of Tor.

Note that the socket mismanagement problem is not solved simply by parallelizing the libevent notification system and priority scheduling processes (which would require complex code), or by utilizing classful queuing disciplines in the kernel (which would require root privileges); while these may improve control over traffic priority to some extent, they would still result in bloated buffers containing data that cannot be sent due to closed TCP congestion windows.

5.2 The KIST Algorithm

In order to overcome the inefficiencies resulting from Tor's socket management, KIST chooses between *all circuits* that have queued data *irrespective* of the socket to which the circuit belongs, and *dynamically adjusts the amount written* to each socket based on real-time kernel information. We now detail each of these approaches.

[6]The small-scale experiments described here are meant to isolate Tor's internal queuing behavior for analysis purposes, and do not fully represent the live Tor network, its background traffic, or its load.

Algorithm 1 The KIST `NotifySocketWritable()` callback, invoked by libevent for each writable socket.

Require: $sdesc, conn, \mathscr{T} \leftarrow GlobalWriteTimeout$

1: $L_p \leftarrow getPendingConnectionList()$
2: **if** L_p **is** $Null$ **then**
3: $\quad L_p \leftarrow newList()$
4: $\quad setPendingConnectionList(L_p)$
5: $\quad createCallback(\mathscr{T}, \texttt{NotifyGlobalWrite()})$
6: **end if**
7: **if** $L_p.contains(conn)$ **is** $False$ **then**
8: $\quad L_p.add(conn)$
9: **end if**
10: $disableNotify(sdesc)$

Global Circuit Scheduling: Recall that libevent delivers write notification events for a single socket at a time. Our approach with KIST is relatively straightforward: rather than handle the kernel write task immediately when libevent notifies Tor that a socket is writable, we simply collect a set of sockets that are writable over a time interval specified by an adjustable `GlobalWriteTimeout` parameter. This allows us to increase the number of candidate circuits we consider when scheduling and writing cells to the kernel: we may select among all circuits which contain cells that are waiting to be written to one of the sockets in our writable set.

The socket collection approach is outlined in Algorithm 1. The socket descriptor `sdesc` and a connection state object `conn` are supplied by libevent. Note that we disable notification events for the socket (as shown in line 10) in order to prevent duplicate notification events during the socket collection interval.

After the `GlobalWriteTimeout` time interval, KIST begins writing cells to the sockets according to the circuit scheduling policy. There are two major phases to this process, which is outlined in Algorithm 2. In lines 4 and 8, we distinguish sockets that contain raw bytes ready to be written directly to the kernel (previously scheduled cells with TLS headers attached) from those with additional cells ready to be converted to raw bytes. KIST first writes the already scheduled raw bytes (lines 4-7), and then schedules and writes additional cells after converting them to raw bytes and adding TLS headers (lines 13-15). Note that the connections should be enumerated (on line 3 of Algorithm 2) in an order that respects the order in which cells were converted to raw bytes by the circuit scheduler in the previous round.

The global scheduling approach does not by itself solve the bloated socket buffer problem. KIST also dynamically computes socket write limits on line 2 of Algorithm 2 using real-time TCP, socket, and bandwidth information, which it then uses when deciding how much to write to the kernel.

Algorithm 2 The KIST `NotifyGlobalWrite()` callback, invoked after the `GlobalWriteTimeout` period.

1: $L_{eligible} \leftarrow newList()$
2: $K \leftarrow collectKernelInfo(getConnectionList())$
3: **for all** *conn* **in** $getPendingConnectionList()$ **do**
4: **if** $hasBytesForKernel(conn)$ **is** $True$ **then**
5: $enableNotify(conn)$
6: $nBytes \leftarrow writeBytesToKernel(K, conn)$
7: **end if**
8: **if** $hasCells(conn)$ **is** $True$ **and** $getLimit(K, conn) > 0$ **then**
9: $L_{eligible}.add(conn)$
10: **end if**
11: **end for**
12: **while** $L_{eligible}.isEmpty()$ **is** $False$ **do**
13: $conn \leftarrow scheduleCell(L_{eligible})$ {cell to bytes}
14: $enableNotify(conn)$
15: $nBytes \leftarrow writeBytesToKernel(K, conn)$
16: **if** $nBytes$ **is** 0 **or** $getLimit(K, conn)$ **is** 0 **then**
17: $L_{eligible}.remove(conn)$
18: **end if**
19: **end while**

Managing Socket Output: KIST attempts to move the queuing delays from the kernel outbound queue to Tor's circuit queue by keeping kernel output buffers as small as possible, i.e., by only writing to the kernel as much as the kernel will actually send. By delaying the circuit scheduling decision until the last possible instant before kernel starvation occurs, Tor will ultimately improve its control over the priority of outgoing data. This approach attempts to give Tor approximately the same control over outbound data that it would have if it had direct access to the network interface. When combined with global circuit scheduling, Tor's influence over outgoing data priority should improve.

To compute write limits, KIST first makes three system calls for each connection: `getsockopt` on level `SOL_SOCKET` for option `SO_SNDBUF` to get *sndbufcap*, the capacity of the send buffer; `ioctl` with command `SIOCOUTQ` to get *sndbuflen*, the current length of the send buffer; and `getsockopt` on level `SOL_TCP` for option `TCP_INFO` to get *tcpi*, a variety of TCP state information. The TCP information used by KIST includes the connection's maximum segment size *mss*, the congestion window *cwnd*, and the number of *unacked* packets for which the kernel is waiting for an acknowledgment from the TCP peer. KIST then computes a write limit for each connection c as follows:

$$\begin{aligned} socket\_space_c &= sndbufcap_c - sndbuflen_c \\ tcp\_space_c &= (cwnd_c - unacked_c) \cdot mss_c \\ limit_c &= \min(socket\_space_c, tcp\_space_c) \end{aligned} \tag{1}$$

The key insight in Equation 1 is that TCP will not allow the kernel to send more packets than dictated by the congestion window, and that the unacknowledged packets prevent the congestion window from sliding open. By respecting this write limit for each connection, KIST ensures that the data sent to the kernel is immediately sendable and reduces kernel queuing delays.

If all connections are sending data in parallel, it is still possible to overwhelm the kernel with more data than it can physically send to the network. Therefore, KIST also computes a global write limit at the beginning of each `GlobalWriteTimeout` period:

$$\begin{aligned} sndbuflen\_prev &= sndbuflen \\ sndbuflen &= \textstyle\sum_{c_i} (sndbuflen_{c_i}) \\ bytes\_sent &= sndbuflen - sndbuflen\_prev \\ limit &= \max(limit, bytes\_sent) \end{aligned} \tag{2}$$

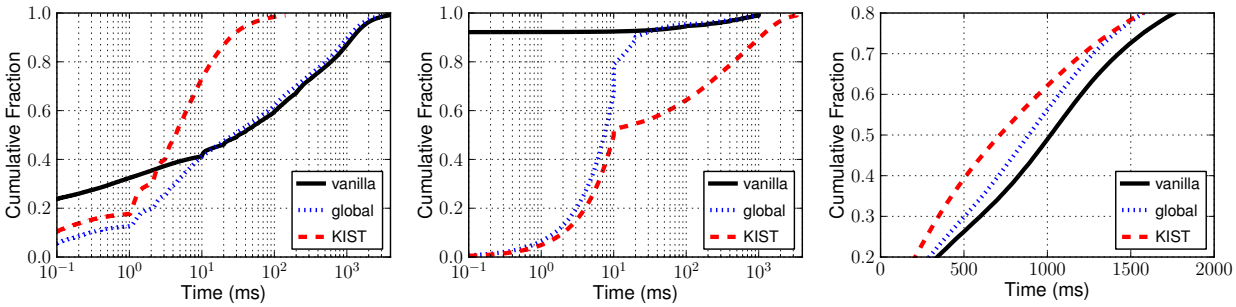
Note that Equation 2 is an attempt to measure the actual upstream bandwidth speed of the machine. In practice, this could be done in a testing phase during which writes are not limited, configured manually, or estimated using other techniques such as packet trains [32].

The connection and global limits are computed at the beginning of a scheduling round, i.e., on line 2 of Algorithm 2; they are enforced whenever bytes are written to the kernel, i.e., on lines 6 and 15 of Algorithm 2. Note that they will be bounded above by Tor's independently configured connection and global application rate limits.

5.3 Experiments and Results

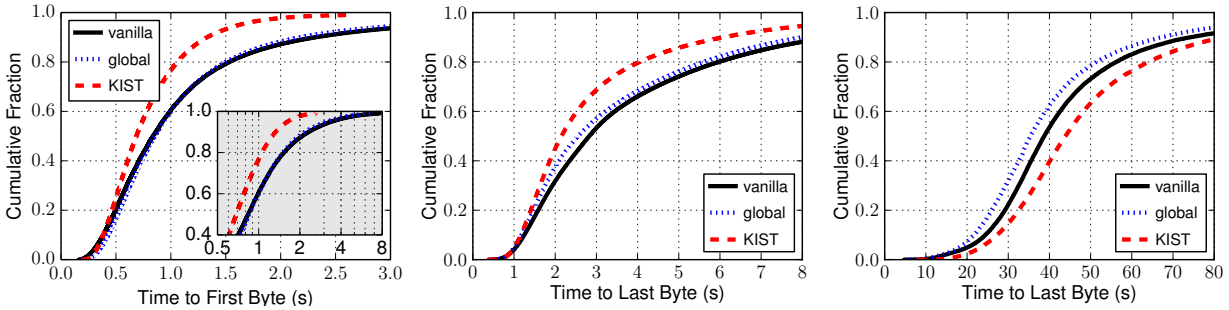
We use Shadow and its models as discussed in Section 3 to measure KIST's effect on network performance, congestion, and throughput. We also evaluate its CPU overhead. See Appendix C [34] for an analysis under a more heavily loaded Shadow-Tor network. Note that we found that KIST performs as well or better under heavier load than under normal load as presented in this section, indicating that it can gracefully scale as Tor grows.

Prototype: We implemented a KIST protoype as a patch to Tor version `0.2.3.25`, and included the elements discussed in Section 4 necessary for measuring congestion during our experiments. We tested vanilla Tor using the default `CircuitPriorityHalflife` of 30, the global scheduling part of KIST (without enforcing the write limits), and the complete KIST algorithm. We configured the global scheduler to use a 10 millisecond `GlobalWriteTimeout` in both the global and KIST experiments. Note that our KIST implementation ignores the connection enumeration order on line 3 of Algorithm 2, an optimization that may further improve Tor's control over priority in cases where the global limit is reached before the algorithm reaches line 12.



(a) Kernel Out Congestion (b) Tor Congestion (c) Circuit Congestion

Figure 6: Congestion for vanilla Tor, KIST, and the global scheduling part of KIST (without enforcing write limits). Figures 6a and 6b show the distribution of cell congestion local to each relay (with logarithmic x-axes), while Figure 6c shows the distribution of the end-to-end circuit congestion for all measured cells.



(a) All Clients (b) 320 KiB "web" clients (c) 5 MiB "bulk" clients

Figure 7: Client performance for vanilla Tor, KIST, and the global scheduling part of KIST (without enforcing write limits). Figure 7a shows the distribution of the time until the client receives the first byte of the data payload, for all clients, while the inset graph shows the same distribution with a logarithmic x-axis. Figures 7b and 7c show the distribution of time to complete a 320 KiB and 5 MiB file by the "web" and "bulk" clients, respectively.

Congestion: Recall that the goal of KIST is to move congestion from the kernel outbound queue to Tor where it can better be managed. Figure 6 shows KIST's effectiveness in this regard. In particular, Figure 6a shows that KIST reduces kernel outbound congestion over vanilla Tor by one to two order of magnitude for over 40 percent of the sampled cells. Further, it shows that the queue time is less than 200 milliseconds for 99 percent of the cells measured, compared to over 4000 milliseconds for both vanilla Tor and global scheduling alone.

Figure 6b shows how global scheduling and KIST increase the congestion inside of Tor. Both global scheduling and KIST result in sharp Tor queue time increases up to 10 milliseconds, after which the existing 10 millisecond `GlobalWriteTimeout` timer event will fire and Tor will flush more data to the kernel. With global scheduling, most of the data queued in Tor quickly gets transferred to the kernel following this timeout, whereas data is queued inside of Tor much longer when using KIST. This result is an explicit feature of KIST, as it means Tor will have more control over data priority when scheduling circuits.

While we have shown above how KIST is able to move congestion from the kernel into Tor, Figure 6c shows the aggregate effect on cell congestion during its complete existence through the entire end-to-end circuit. KIST reduces aggregate circuit congestion from 1010.1 milliseconds to 704.5 milliseconds in the median, a 30.3 percent improvement, while global scheduling reduces congestion by 13 percent to 878.8 milliseconds.

The results in Figure 6 show that KIST indeed achieves its congestion management goals while highlighting the importance of limiting kernel write amounts in addition to globally scheduling circuits.

Performance: We show in Figure 7 how KIST affects client performance. Figure 7a shows how network latency is generally affected by showing the time until the first byte of every download by all clients. Global scheduling alone is roughly indistinguishable from vanilla Tor, while KIST reduces latency to the first byte for over 80 percent of the downloads—in the median, KIST reduces network latency by 18.1 percent from 0.838 seconds to 0.686 seconds. The inset graph has a logarithmic x-axis and shows that KIST is particularly

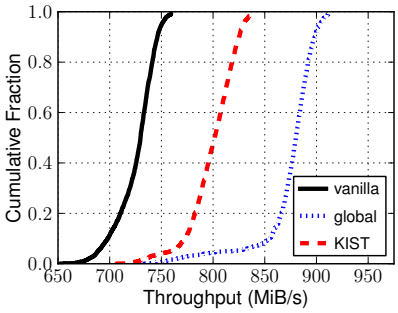


Figure 8: Aggregate relay write throughput for vanilla Tor, KIST, and the global scheduling part of KIST (without enforcing write limits). Both of our enhancements increase network throughput over vanilla Tor.

beneficial in the upper parts of the distribution: in the 99th percentile, latency is reduced from more than 7 seconds to less than 2.7 seconds.

Figures 7b and 7c show the distribution of time to complete each 320 KiB download for the "web" clients and each 5 MiB file for the "bulk" clients, respectively. In our experiments, the 320 KiB download times decreased by over 1 second for over 40 percent of the downloads, while the download times for 5 MiB files increased by less than 8 seconds for all downloads. These changes in download times are a result of Tor correctly utilizing its circuit priority scheduler, which prioritizes traffic with the lowest exponentially-weighted moving average throughput. As the "web" clients pause between downloads, their traffic is often prioritized ahead of "bulk" traffic. Our results indicate that not only does KIST decrease Tor network latency, it also increases Tor's ability to appropriately manage its traffic.

Throughput: We show in Figure 8 KIST's effect on relay throughput. Shown is the distribution of aggregate bytes written per second by all relays in the network. We found that throughput improves when using KIST due to a combination of the reduction in network latency and our client model: web clients completed their downloads faster in the lower latency network and therefore also downloaded more files. By lowering circuit congestion, KIST improves utilization of existing bandwidth resources over vanilla Tor by 71.6 MiB/s, or 9.8%, in the median. While the best network utilization is achieved with global scheduling without write limits (a 150.1 MiB/s, or 20.5%, improvement over vanilla Tor in the median), we have shown above that it is less effective than KIST at reducing kernel congestion and allowing Tor to correctly prioritize traffic.

Overhead: The main overhead in KIST involves the collection of socket and TCP information from the kernel using three separate calls to `getsockopt` (socket capacity, socket length, and TCP info). These three system calls are made for every connection after every `GlobalWriteTimeout` interval. To understand the overhead involved with these calls, we instrumented curiosity3 from Section 4 to collect timing information while performing the syscalls required by KIST. Our test ran on the live relay running an Intel Xeon x3450 CPU at 2.67GHz for 3 days and 14 hours, and collected a timing sample every second for a total of 309,739 samples. We found that the three system calls took 0.9140 microseconds per connection in the median, with a mean of 0.9204 and a standard deviation of 3.1×10^{-5}.

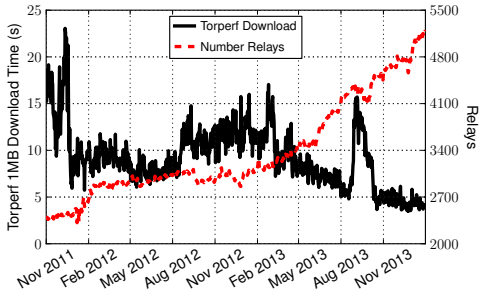


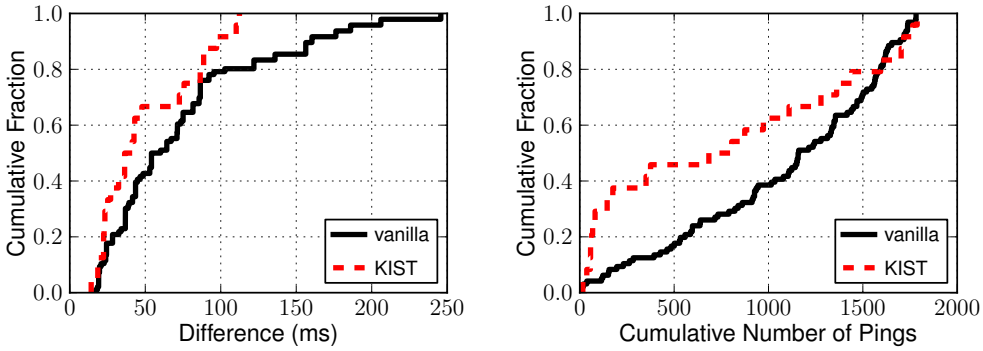
Figure 9: Network size correlates with performance.

The number of connections a relay may have is bounded above roughly by the number of relays in the network, which is currently around 5,000. Therefore, we expect the overhead to be less than 5 milliseconds and reasonable for current relays. If this overhead becomes problematic as Tor grows, the gathering of kernel information can be outsourced to a helper thread and continuously updated over time. Further, we have determined through discussions with Linux kernel developers that the *netlink socket diag* interface could be used to collect information for several sockets at once—an optimization that may provide significant reductions in overhead.

6 Security Analysis

Performance and Security: Performance and ease of use affect adoption rates of any network technology. They have played a central role in the size and diversity of the Tor userbase. This can then affect the size of the network itself as users are more willing to run parts of the network or contribute financially to its upkeep, e.g., via torservers.net. Growth from performance improvements affect the security of Tor by increasing the uncertainty for many types of adversaries concerning who is communicating with whom [17, 20, 22, 37]. Performance factors in anonymous communication systems like Tor are thus pertinent to security in a much more direct way than they typically would be for, say, a faster signature algorithm's impact on the security of an authentication system.

Though real and more significant, direct effects of performance on Tor's security from network and userbase growth are also hard to show, given both the variety of



(a) Latency Estimate Difference

(b) Cumulative Pings until Best Estimate

Figure 10: Latency leaks are more pronounced (10a) and are faster (10b) with KIST.

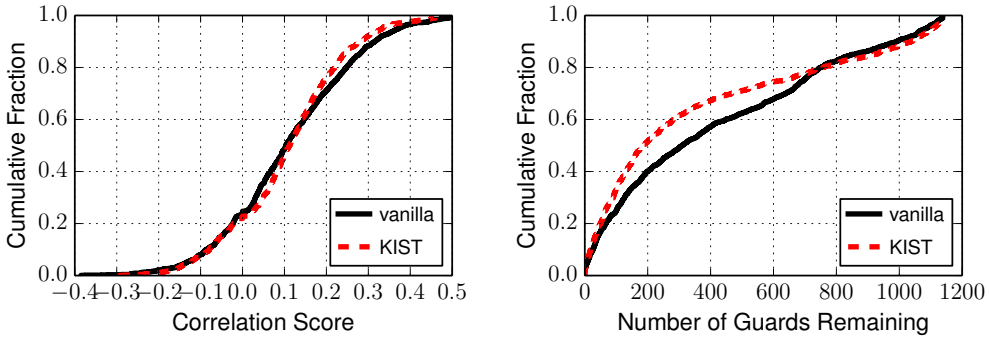
causal factors and the difficulty of gathering useful data while preserving privacy. Whatever the causal explanation, a correlation ($-.62$) between Tor performance improvement over time (measured by the median download time of a 1 MB file) and network size is shown in Figure 9. (Numbers are from the Tor Metrics Portal [8].) Similar results hold when number of relays is replaced with bandwidth metrics. Which is more relevant depends on the adversary's most significant constraints: adversary size and distribution across the underlying network are important considerations [39].

More measurable effects can occur if a performance change creates a new opportunity for attack or makes an existing attack more effective or easier to mount. Performance change may also eliminate or diminish previous possible or actual attacks. Growth effects are potentially the greatest security effects of our performance changes, but we now focus on these more directly observable aspects. They include attacks on Tor based on resource contention or interference [19, 24, 25, 31, 44, 46, 48] or simply available resource observation [44], or observing other performance properties, such as latency [31]. Many papers have also explored improvements to Tor performance via scheduling, throttling, congestion management, etc. (see Section 2). Manipulating performance enhancement mechanisms can turn them into potential vectors of attack themselves [38]. Geddes *et al.* [25] analyzed anonymity impact of several performance enhancement mechanisms for Tor.

Latency Leak: The basic idea of a latency leak attack as first set out in [30] is to measure RTT (roundtrip time) between a compromised exit and the client of some target connection repeatedly and then to pick the shortest result as an indication of latency. Next compare this to the known, measured latency through all the hops in the circuit except client to entry relay. (Other attacks such as throughput measurement, discussed below, are assumed to have already identified the relays in the circuit.) Next, use that to determine the latency between the client of the target connection and its entry relay, which is in a known network location. This can significantly reduce the range of possible network locations for the client. When measuring latency using our improved models and simulator, we discovered that this attack is generally able to determine latency well with vanilla Tor. While KIST improves the overall accuracy, the improvement is small when a good estimate was also found with vanilla Tor.

Figure 10a shows the results of an experiment run on our model from Section 3 with random circuit and client choices, indicating the difference between the correct latency and the estimate after a few hundred pings once per second. Roughly 20% of circuits for both vanilla Tor and KIST are within 25ms of the correct latency. After this they diverge, but both have a median latency estimate of about 50ms or less. It is only for the worst 10-20% of estimates, which are presumably not useful anyway, that KIST is substantially better. While the eventual accuracy of the attack is comparable for both, the attacker under KIST is significantly faster on average. Figure 10b shows the cumulative number of pings (seconds) until a best estimate is achieved. After 200 pings, nearly 40% of KIST circuits have achieved their best estimate while less than 10% have for vanilla Tor. And the median number of pings needed for KIST is about 700 vs. 1200 for vanilla Tor.

The accuracy of the latency attack indicated above is a significant threat to network location, which from a technical perspective is what Tor is primarily designed to protect. It could be diminished by padding latency. Specifically any connection at the edges of the Tor network, at either source or destination end, could be dynamically padded by the entry or exit relay respectively to ideally make latency of all edge connections through that relay uniform—more realistically to significantly decrease the network location information leaked by latency. Relays can do their own RTT measurements for any edge



(a) Throughput Correlation, Probe to Target Client (b) Guards Remaining in Candidate Guard Set

Figure 11: While the aggregate throughput correlations of the probe to the true guard over the set of all probes (11a) are not significantly affected by KIST, it is slightly easier for the adversary to eliminate candidate guards of a target client (for all clients) (11b) when using KIST.

connection and pad accordingly. There are many issues around this suggestion, which we leave to future work.

Throughput Leak: The throughput attack introduced by Mittal *et al.* [44] identifies the entry relay of a target connection by setting up one-hop probe circuits from attack clients through all prospective entry relays and back to the client in order to measure throughput at those relays. These throughputs are compared to the throughput observed by the exit relay of the target circuit. The attack is directed against circuits used for bulk downloading since these will attempt a sustained maximum throughput, and will result in congestion effects on bottleneck relays that allow the adversary to reduce uncertainty about possible entry relays. Mittal *et al.* also looked at attacks on lower bandwidth interactive traffic and found some success, although with much less accuracy than for bulk traffic.

We analyze the extent to which KIST affects the throughput attack. While measuring throughput at entry relays, we also adopt the simplification of Geddes *et al.* [25] of restricting observations to entry relays that are not used as middle or exit relays for other bulk downloading circuits. This allows us the efficiency of making measurements for several simulated attacks simultaneously while minimizing interference between their probes.

Figure 11a shows the cumulative distribution of scores for correlation of probe throughput at the correct entry relay with throughput at the observed exit relay under vanilla Tor and under KIST scheduling (on the network and user model given in Section 3). Throughput was measured every 100 ms. We found that the throughput correlations are not significantly affected by KIST.

To explain the correlation scores, recall from Section 5.2 how KIST reduces both circuit congestion and network latency by allowing Tor to properly prioritize circuits independent of the TCP connections to which they belong. This leads to two competing potential effects on the throughput attack: (1) a less congested network will increase the sensitivity of the probes to variations in throughput, thereby allowing stronger correlations between the throughput achieved by the probe client and that achieved by the target client; and (2) a circuit's throughput is most correlated with that of its bottleneck relay, and KIST's improved scheduling should also reduce the bottleneck effects of congestion in the network and allow weaker throughput correlations. Further, the improved priority scheduling (moving from round-robin over TCP connections to properly utilizing EWMA over circuits) will cause the throughput of each client to become slightly "burstier" over the short term as the priority causes the scheduler to oscillate between the circuits. We suspect that the similar correlation scores are the result of combining these effects.

To further understand KIST's affect on the throughput attack, we measure how the correlation of every client's throughput to the *true* guard's throughput compares to the correlation of the client's throughput to that of every other *candidate* guard in the network. For every client, we start with a candidate guard set of all guards, and remove those guards with a lower correlation score with the client than the true guard's score. Figure 11b shows the distribution, over all clients, of the extent to which we were able to reduce the size of the candidate guard set using this heuristic. Although KIST reduced the uncertainty about the true guard used by the target client, we do not expect the small improvement to significantly affect the ability to conduct a successful throughput attack in practice.

7 Conclusion

In this paper, we outlined the results of an in-depth congestion study using both public and private Tor net-

works. We identified that most congestion occurs in outbound kernel buffers, analyzed Tor socket management, and designed a new socket transport mechanism called KIST. Through evaluation in a full-scale private Shadow-Tor network, we conclude that KIST is capable of moving congestion into Tor where it can be better managed by application priority scheduling mechanisms. More specifically, we found that by considering all sockets and respecting TCP state information when writing data to the kernel, KIST reduces both congestion and latency while increasing utilization. Finally, we performed a detailed evaluation of KIST against well-known latency and throughput attacks. While KIST increases the speed at which true network latency can be calculated, it does not significantly affect the accuracy of the probes required to correlate throughput.

Future work should extend our simulation-based evaluation and consider how KIST performs for relays in the live Tor network. We note that our analysis is based exclusively on Linux relays, as 91% of Tor's bandwidth is provided by relays running a Linux-based distribution [58]. Although we expect KIST to improve performance similarly across platforms because it primarily works by managing socket buffer levels, future work should consider how KIST is affected by the interoperation of relays running on a diverse set of OSes. Finally, our KIST prototype would benefit from optimizations, particularly by running the process of gathering kernel state information in a separate thread and/or using the *netlink socket diag* interface.

Acknowledgments

We thank our shepherd, Rachel Greenstadt, and the anonymous reviewers for providing feedback that helped improve this work. We thank Roger Dingledine for discussions about measuring congestion in Tor, and Patrick McHardy for suggesting the use of the netlink socket diag interface. This work was partially supported by ONR, DARPA, and the National Science Foundation through grants CNS-1149832, CNS-1064986, CNS-1204347, CNS-1223825, and CNS-1314637. This material is based upon work supported by the Defense Advanced Research Project Agency (DARPA) and Space and Naval Warfare Systems Center Pacific under Contract No. N66001-11-C-4020. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Defense Advanced Research Project Agency and Space and Naval Warfare Systems Center Pacific.

References

[1] Alexa top 1 million sites. http://s3.amazonaws.com/alexa-static/top-1m.csv.zip. Retrieved 2012-01-31.

[2] CAIDA data. http://www.caida.org/data.

[3] DETER testbed. http://www.isi.edu/deter.

[4] libevent event notification library. http://libevent.org/.

[5] libkqtime code repository. https://github.com/robgjansen/libkqtime.git.

[6] libpcap portable C/C++ library for network traffic capture. http://www.tcpdump.org/.

[7] Shadow homepage and code repositories. https://shadow.github.io/, https://github.com/shadow/.

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

[9] TorPerf. https://gitweb.torproject.org/torperf.git/.

[10] TorPS homepage. http://torps.github.io/.

[11] AKHOONDI, M., YU, C., AND MADHYASTHA, H. V. LASTor: A low-latency as-aware Tor client. In *IEEE Symposium on Security and Privacy (Oakland)* (2012).

[12] ALLMAN, M., PAXSON, V., AND BLANTON, E. TCP Congestion Control. RFC 5681 (Draft Standard), Sept. 2009.

[13] ALSABAH, M., BAUER, K., ELAHI, T., AND GOLDBERG, I. The path less travelled: Overcoming Tor's bottlenecks with traffic splitting. In *Privacy Enhancing Technologies Symposium (PETS)* (2013).

[14] ALSABAH, M., BAUER, K., AND GOLDBERG, I. Enhancing Tor's performance using real-time traffic classification. In *ACM Conference on Computer and Communications Security (CCS)* (2012).

[15] ALSABAH, M., BAUER, K., GOLDBERG, I., GRUNWALD, D., MCCOY, D., SAVAGE, S., AND VOELKER, G. DefenestraTor: Throwing out windows in Tor. In *Privacy Enhancing Technologies Symposium (PETS)* (2011).

[16] ALSABAH, M., AND GOLDBERG, I. PCTCP: Per-circuit tcp-over-ipsec transport for anonymous communication overlay networks. In *ACM Conference on Computer and Communications Security (CCS)* (2013).

[17] BACK, A., MÖLLER, U., AND STIGLIC, A. Traffic analysis attacks and trade-offs in anonymity providing systems. In *Workshop on Information Hiding (IH)* (2001).

[18] CHAABANE, A., MANILS, P., AND KAAFAR, M. Digging into anonymous traffic: A deep analysis of the tor anonymizing network. In *IEEE Conference on Network and System Security (NSS)* (2010).

[19] CHAN-TIN, E., SHIN, J., AND YU, J. Revisiting circuit clogging attacks on Tor. In *IEEE Conference on Availability, Reliability and Security (ARES)* (2013).

[20] DINGLEDINE, R., AND MATHEWSON, N. Anonymity loves company: Usability and the network effect. In *Workshop on the Economics of Information Security (WEIS)* (2006).

[21] DINGLEDINE, R., MATHEWSON, N., AND SYVERSON, P. Tor: The second-generation onion router. In *USENIX Security Symposium (USENIX)* (2004).

[22] DINGLEDINE, R., MATHEWSON, N., AND SYVERSON, P. Deploying low-latency anonymity: Design challenges and social factors. *IEEE Security & Privacy 5*, 5 (Sept./Oct. 2007), 83–87.

[23] DINGLEDINE, R., AND MURDOCH, S. J. Performance improvements on Tor or, why Tor is slow and what we're going to do about it. Tech. Rep. 2009-11-001, The Tor Project, 2009.

[24] EVANS, N. S., DINGLEDINE, R., AND GROTHOFF, C. A practical congestion attack on Tor using long paths. In *USENIX Security Symposium (USENIX)* (2009).

[25] GEDDES, J., JANSEN, R., AND HOPPER, N. How low can you go: Balancing performance with anonymity in Tor. In *Privacy Enhancing Technologies Symposium (PETS)* (2013).

[26] GOPAL, D., AND HENINGER, N. Torchestra: Reducing interactive traffic delays over Tor. In *ACM Workshop on Privacy in the Electronic Society (WPES)* (2012).

[27] HA, S., RHEE, I., AND XU, L. CUBIC: a new TCP-friendly high-speed TCP variant. *ACM SIGOPS Operating Systems Review 42*, 5 (2008), 64–74.

[28] HAHN, S., AND LOESING, K. Privacy-preserving ways to estimate the number of Tor users. Tech. Rep. 2010-11-001, The Tor Project, 2010.

[29] HOPPER, N. Protecting Tor from botnet abuse in the long term. Tech. Rep. 2013-11-001, The Tor Project, 2013.

[30] HOPPER, N., VASSERMAN, E. Y., AND CHAN-TIN, E. How much anonymity does network latency leak? In *ACM Conference on Computer and Communications Security (CCS)* (2007). Expanded and revised version in [31].

[31] HOPPER, N., VASSERMAN, E. Y., AND CHAN-TIN, E. How much anonymity does network latency leak? *ACM Transactions on Information and System Security (TISSEC) 13*, 2 (Feb. 2010), 13–28.

[32] JAIN, R., AND ROUTHIER, S. Packet trains–measurements and a new model for computer network traffic. *IEEE Selected Areas in Communications 4*, 6 (1986), 986–995.

[33] JANSEN, R., BAUER, K., HOPPER, N., AND DINGLEDINE, R. Methodically modeling the Tor network. In *USENIX Workshop on Cyber Security Experimentation and Test (CSET)* (2012).

[34] JANSEN, R., GEDDES, J., WACEK, C., SHERR, M., AND SYVERSON, P. Appendices to accompany "Never been KIST: Tor's congestion management blossoms with kernel-informed socket transport". Tech. Rep. 14-012, Univ. of Minnesota, 2014. http://www.cs.umn.edu/tech_reports_upload/tr2014/14-012.pdf.

[35] JANSEN, R., AND HOPPER, N. Shadow: Running Tor in a box for accurate and efficient experimentation. In *USENIX Security Symposium (USENIX)* (2012).

[36] JANSEN, R., HOPPER, N., AND KIM, Y. Recruiting new Tor relays with BRAIDS. In *ACM Conference on Computer and Communications Security (CCS)* (2010).

[37] JANSEN, R., JOHNSON, A., AND SYVERSON, P. LIRA: Lightweight incentivized routing for anonymity. In *Network and Distributed System Security Symposium (NDSS)* (2013).

[38] JANSEN, R., SYVERSON, P., AND HOPPER, N. Throttling Tor bandwidth parasites. In *USENIX Security Symposium (USENIX)* (2012).

[39] JOHNSON, A., WACEK, C., JANSEN, R., SHERR, M., AND SYVERSON, P. Users get routed: Traffic correlation on tor by realistic adversaries. In *ACM Conference on Computer and Communications Security (CCS)* (2013).

[40] MATHEWSON, N. Evaluating SCTP for Tor. http://archives.seul.org/or/dev/Sep-2004/msg00002.html, Sept. 2004. Listserv posting.

[41] MATHIS, M., AND MAHDAVI, J. Forward acknowledgement: Refining TCP congestion control. *ACM SIGCOMM Computer Communication Review 26*, 4 (1996), 281–291.

[42] MATHIS, M., MAHDAVI, J., FLOYD, S., AND ROMANOW, A. TCP Selective Acknowledgment Options. RFC 2018 (Proposed Standard), Oct. 1996.

[43] MCCOY, D., BAUER, K., GRUNWALD, D., KOHNO, T., AND SICKER, D. Shining light in dark places: Understanding the Tor network. In *Privacy Enhancing Technologies Symposium (PETS)* (2008).

[44] MITTAL, P., KHURSHID, A., JUEN, J., CAESAR, M., AND BORISOV, N. Stealthy traffic analysis of low-latency anonymous communication using throughput fingerprinting. In *ACM Conference on Computer and Communications Security (CCS)* (2011).

[45] 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 *Annual Computer Security Applications Conference (ACSAC)* (2011).

[46] MURDOCH, S. J. Hot or not: Revealing hidden services by their clock skew. In *ACM Conference on Computer and Communications Security (CCS)* (2006).

[47] MURDOCH, S. J. Comparison of Tor datagram designs. Tech. Rep. 2011-11-001, The Tor Project, 2011.

[48] MURDOCH, S. J., AND DANEZIS, G. Low-cost traffic analysis of Tor. In *IEEE Symposium on Security and Privacy (Oakland)* (2005).

[49] NOWLAN, M. F., TIWARI, N., IYENGAR, J., AMIN, S. O., AND FORD, B. Fitting square pegs through round pipes. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2012).

[50] NOWLAN, M. F., WOLINSKY, D., AND FORD, B. Reducing latency in Tor circuits with unordered delivery. In *USENIX Workshop on Free and Open Communications on the Internet (FOCI)* (2013).

[51] The ns2 Network Simulator. http://www.isi.edu/nsnam/ns/.

[52] PAXSON, V., ALLMAN, M., CHU, J., AND SARGENT, M. Computing TCP's Retransmission Timer. RFC 6298 (Proposed Standard), June 2011.

[53] RAMACHANDRAN, S. Web metrics: Size and number of resources. https://developers.google.com/speed/articles/web-metrics, 2010.

[54] REARDON, J., AND GOLDBERG, I. Improving Tor using a TCP-over-DTLS tunnel. In *USENIX Security Symposium (USENIX)* (2009).

[55] SHERR, M., MAO, A., MARCZAK, W. R., ZHOU, W., LOO, B. T., AND BLAZE, M. A3: An Extensible Platform for Application-Aware Anonymity. In *Network and Distributed System Security Symposium (NDSS)* (2010).

[56] SNADER, R., AND BORISOV, N. A tune-up for Tor: Improving security and performance in the Tor network. In *Network and Distributed System Security Symposium (NDSS)* (2008).

[57] TANG, C., AND GOLDBERG, I. An improved algorithm for Tor circuit scheduling. In *ACM Conference on Computer and Communications Security (CCS)* (2010).

[58] Tor network status. http://torstatus.blutmagie.de/index.php.

[59] WACEK, C., TAN, H., BAUER, K., AND SHERR, M. An empirical evaluation of relay selection in Tor. In *Network and Distributed System Security Symposium (NDSS)* (2013).

[60] WANG, T., BAUER, K., FORERO, C., AND GOLDBERG, I. Congestion-aware path selection for Tor. In *Financial Cryptography and Data Security (FC)* (2012).

[61] WEIGLE, E., AND FENG, W.-C. A comparison of TCP automatic tuning techniques for distributed computing. In *IEEE Symposium on High Performance Distributed Computing (HPDC)* (2002).