KIST: Kernel-Informed Socket Transport for Tor

ROB JANSEN and MATTHEW TRAUDT, U.S. Naval Research Laboratory, USA
JOHN GEDDES, University of Minnesota
CHRIS WACEK and MICAH SHERR, Georgetown University, USA
PAUL SYVERSON, U.S. Naval Research Laboratory, USA

Tor's growing popularity and user diversity has resulted in network performance problems that are not well understood, though performance is understood to be a significant factor in Tor's security. A large body of work has attempted to solve performance problems without a complete understanding of where congestion occurs in Tor. In this article, 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 contributors to Tor's congestion: Tor writes sockets sequentially, and Tor writes as much as possible to each socket. To improve Tor's performance, we 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 circuits of all writable sockets* when scheduling cells. We find that, in the medians, KIST reduces circuit congestion by more than 30%, reduces network latency by 18%, and increases network throughput by nearly 10%. We also find that client and relay performance with KIST improves as more relays deploy it and as network load and packet loss rates increase. We analyze the security of KIST and find an acceptable performance and security tradeoff, as it does not significantly affect the outcome of well-known latency, throughput, and traffic correlation attacks. KIST has been merged and configured as the default socket scheduling algorithm in Tor version `0.3.2.1-alpha` (released September 18, 2017) and became stable in Tor version `0.3.2.9` (released January 9, 2018). 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.

This work has been partially supported by the Office of Naval Research (ONR), the Defense Advanced Research Project Agency (DARPA), the National Science Foundation (NSF) under grant numbers CNS-1149832, CNS-1064986, CNS-1204347, CNS-1223825, CNS-1314637, and CNS-1527401, and the Department of Homeland Security (DHS) Science and Technology Directorate, Homeland Security Advanced Research Projects Agency, Cyber Security Division under agreement number FTCY1500057. The views expressed in this work are strictly those of the authors and do not necessarily reflect the official policy or position of any funding agency. 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.

Authors' addresses: R. Jansen, M. Traudt, and P. Syverson, U.S. Naval Research Laboratory, Washington, DC, USA; emails: {rob.g.jansen, matthew.traudt, paul.syverson}@nrl.navy.mil; J. Geddes, University of Minnesota, Minneapolis, MN, USA; email: geddes@cs.umn.edu; C. Wacek and M. Sherr, Georgetown University, Washington, DC, USA; emails: {wacek, msherr}@cs.georgetown.edu.

ACM 2471-2566/2018/12-ART3 $15.00
https://doi.org/10.1145/3278121

CCS Concepts: • **Security and privacy → Pseudonymity, anonymity and untraceability**; **Distributed systems security**; • **Networks → Network performance evaluation**; **Network privacy and anonymity**; *Transport protocols*; • **Computing methodologies** → Modeling methodologies; Discrete-event simulation;

Additional Key Words and Phrases: Anonymous communication, Tor performance

ACM Reference format:
Rob Jansen, Matthew Traudt, John Geddes, Chris Wacek, Micah Sherr, and Paul Syverson. 2018. KIST: Kernel-Informed Socket Transport for Tor. *ACM Trans. Priv. Secur.* 22, 1, Article 3 (December 2018), 37 pages.
https://doi.org/10.1145/3278121

1 INTRODUCTION

Tor [14] is the most popular overlay network for communicating anonymously online. Tor serves millions of users daily [44] by transferring their traffic through a source-routed *circuit* of three volunteer relays, and it 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 onion 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 large anonymity set, they have also led to performance issues [16]. For example, it has been shown that roughly a quarter to a half of Tor's traffic can be attributed to BitTorrent [11, 34, 48], while the more recent use of Tor by a botnet [26] has further increased concern about Tor's ability to utilize volunteer resources to handle a growing user base [13, 22, 33, 35, 37, 51].

Numerous proposals have been made to battle Tor's performance problems, some of which modify the mechanisms used for path selection [4, 67, 69], client throttling [5, 38, 51], circuit scheduling [65], and flow/congestion control [6, 66]. 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 article, 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 its continuing growth.

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 we 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 [32], and we 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]

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 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 detect, prevent, or otherwise manage kernel congestion.

[1]We have contributed our patches to the open source Shadow project and they have been integrated as of Shadow release version `1.9.0` (https://shadow.github.io/ and https://github.com/shadow/shadow).

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 [32].

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 proposals that suggest increasing the number of sockets between relays [7, 21, 23].

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 chooses from *all* circuits with writable data rather than just those belonging to a single TCP socket. Second, KIST dynamically manages the amount of data written to each socket based on real-time kernel and TCP state information that can be queried from user space. In this way, KIST attempts to reduce the amount of data that exists in the kernel but that cannot be sent and to increase the amount of time that Tor has control over data priority.

We perform in-depth experiments in large, private Shadow-Tor networks of tens of thousands of Tor nodes, and we show how KIST can be used to relocate congestion from the kernel into Tor where it can be properly managed. We find that KIST allows Tor to correctly utilize its circuit priority scheduler, reducing download latency by more than 660 milliseconds, or 23.5%, for interactive traffic streams typically generated by web browsing behaviors. We also evaluate KIST under a range of network load and packet loss models and show that KIST is able to increasingly improve both client and relay performance relative to Tor's default scheduler as both network load and packet loss rates increase. Finally, we evaluate KIST across a range of values of an important KIST parameter to understand its effect on performance, and we evaluate how KIST affects performance when it is incrementally deployed across relays in the Shadow-Tor network.

We also deploy KIST on a fast relay in the public Tor network and confirm that KIST improves client and relay performance in practice. We find that KIST overhead is tolerable: With our suggested parameter settings, the system call overhead scales linearly with the number of relay-to-relay TCP connections with write-pending data and independently of the total number of open sockets. KIST has been merged into Tor version `0.3.2.1-alpha` on September 9, 2017, was marked stable in Tor version `0.3.2.9` on January 9, 2018, and is configured as the default socket scheduling algorithm.

Performance and Security. KIST focuses on improving Tor's performance, which is fundamental to Tor's security in two separate ways.

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

First, it is widely recognized that usability is a security property in general, and the impact of improved performance on usability is recognized to be a significant contributor to Tor's security in particular [13]. Performance is a significant enough usable security property that large-scale real-world attacks have occurred where adversaries intentionally degrade the performance of more secure Internet communication protocols in order to encourage users to switch to less secure alternatives [8]. By improving Tor's performance, KIST causes Tor to be more useful to more users, and anonymity generally increases with the size of the set of users [1].

Second, perception and growth of Tor network resources is affected by resource utilization and load balancing. In Tor, relay operators voluntarily contribute network resources and are keenly aware of how their volunteered contributions are being utilized (this is openly discussed in operator forums). Better resource utilization may improve perception among volunteers, which could lead to new resources from operators who feel that their contributions are not being ignored or undervalued [15].[3] By improving Tor's resource utilization, KIST may lead to network growth, which in turn could further improve Tor's anonymity [13].

While performance is an important contributor, well-known latency, throughput, and traffic correlation attacks also affect Tor's security. Improvements to latency and throughput have the potential to make these attacks more effective, and so we analyze KIST's impact on these 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 [28]. We also show how KIST does not significantly affect an adversary's ability to collect accurate measurements required for the throughput correlation attack [50] when compared to vanilla Tor. Finally, we argue that KIST would not significantly improve traffic correlation attacks, since it has been shown that the accuracy of such attacks already approaches 100% without KIST [42].

2 BACKGROUND AND RELATED WORK

Tor [14] is a volunteer-operated anonymity service used by hundreds of thousands to millions of users daily [25, 44]. 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.

2.1 Handling Tor Traffic

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 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, an HTTP request to acm.org may result in several Tor streams (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 *cell* that is padded to 512 bytes. 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

[3]Numerous threads on the tor-relays mailing list (https://lists.torproject.org/pipermail/tor-relays/) discuss such concerns.

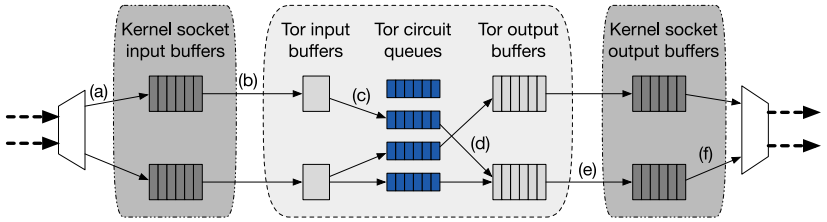


Fig. 1. Tor relay internal cell processing. Dashed lines denote TCP connections. Transitions between buffers—both within the kernel (side boxes) and within Tor (center box)—are shown with solid arrows.

socket input buffers by the underlying OS (Figure 1(a)).[4] 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 1(b)). Upon receipt of an entire TLS datagram, the TLS layer is removed, the cell is onion-encrypted[5] and then transferred and enqueued in the appropriate Tor circuit queue (Figure 1(c)). 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 [65]. The circuit scheduler selects a cell from a circuit queue to process based on this prioritization, onion-encrypts the cell, and stores it in a Tor output buffer (Figure 1(d)). Once the Tor output buffer contains sufficient data to form a TLS packet, the data are written to the kernel for transport (Figure 1(f)).

2.2 Improving Tor Performance

There is a large body of work that attempts to improve Tor's network performance, for example, by refining Tor's relay selection strategy [2, 61, 63] or providing incentives to users to operate Tor relays [22, 33, 35, 37, 51]. 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 article 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 [59], Mathewson explores using SCTP [45], and Tschorsch and Scheuermann design a UDP backpressure-based protocol [66]. Murdoch [53] 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 (to favor interactive web users over bulk downloaders) [65]. Unfortunately, although Tor has adopted EWMA, the network has not significantly benefited from its use [32]. 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.

[4]For simplicity, we consider only relays that run Linux since such relays represent 93% of all Tor relays and contribute 92% of the bandwidth of the live Tor network as of December 4, 2017 ([44]).
[5]Encrypted or decrypted, depending on circuit direction.

AlSabah et al. propose an ATM-like congestion and flow control system for Tor called N23 [6]. 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 uses separate TCP connections to carry interactive and bulk traffic, isolating the effects of congestion between the two traffic classes [23]. Conceptually, Torchestra moves circuit-selection logic to the kernel, where the OS schedules packets for the two connections. Relatedly, AlSabah and Goldberg introduce PCTCP [7], a transport mechanism for Tor in which each circuit is assigned its own TCP socket and IPsec tunnel, and in IMUX [21] Geddes et al. show how the number of sockets between relay pairs can be dynamically tuned. In this article, 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. [56] propose the use of uTCP and uTLS [55] 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 circuits 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 use of the TCP transport mechanism.

3 ENHANCED NETWORK EXPERIMENTATION

We evaluate Tor throughout this article using simulation and, in particular, the Shadow [32] discrete-event network simulation framework. First, we use network simulation because it allows us to easily run experiments over a range of network models and characteristics. It also allows us to run a Tor network *at scale*; that is, with more than 3,000 relays and 10,000 clients. Second, we use Shadow because it is the most widely used Tor experimentation tool [62] and because it allows us to *directly execute* the Tor software (something no other simulator supports, to the best of our knowledge). This allows us to develop our prototypes directly in the Tor software and then simulate, test, and deploy that prototype without implementing it multiple times.

Shadow uses function interposition to intercept all necessary system calls and then redirects them to their simulated counterpart, thereby emulating a Linux operating system to any application it runs. Shadow transparently supports applications that create threads, open UDP and TCP sockets, read and write to sockets, perform blocking system calls, and more. Applications are compiled as position-independent executables and loaded into Shadow as plug-ins at run time, and then they are directly executed for each virtual simulation node that is configured to run it. Shadow's strong support for network-based distributed systems in general and Tor in particular make it ideal for evaluating Tor network attacks and defenses [20, 36, 43] as well as network-wide effects of proposed Tor algorithms [21, 29, 39, 40, 60].

To increase confidence in our experiments, we introduce three significant enhancements to the Shadow simulator [32] and its existing models [31]: a more realistic simulated kernel and TCP network stack, an updated Internet topology model, and validated models of the largest known deployed private Tor network. The enhancements in this section represent a large and determined engineering effort; we will show how simulation 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 among the various research groups around the world that use the simulator.

3.1 Supporting TCP Features

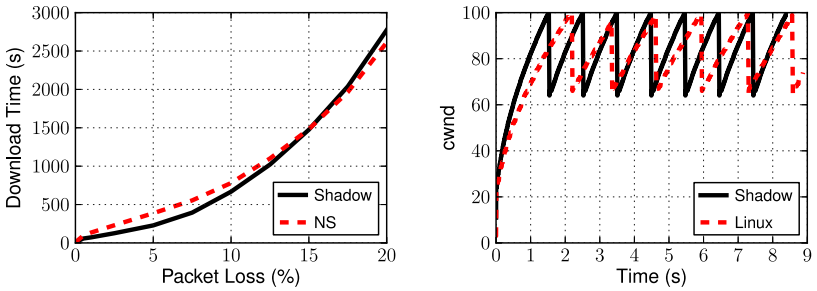
After initially reviewing Shadow's implementation, we discovered that it was missing many important TCP features, causing it to be less accurate than desired. We enhanced Shadow by adding the following: retransmission timers [57], fast retransmit/recovery [3], selective acknowledgments [47], and forward acknowledgments [46]; all are currently used in Linux's TCP implementation. We also 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 [24], 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 briefly discuss the TCP algorithms incorporated into Shadow and show how our implementation of these algorithms greatly enhance Shadow's accuracy, which is paramount to the remainder of this article.

3.1.1 Detecting Packet Loss. We have implemented in Shadow four techniques commonly found in TCP implementations for detecting and handling packet loss. First, we compute the retransmission timer value [57] and set up events to notify Shadow when a packet is lost and should be retransmitted. Second, as part of fast retransmit/recovery [3], once three duplicate acknowledgments have been received, all unacknowledged packets are considered lost and are resent. Third, we use selective acknowledgments [47] so the receiver can notify the sender of any out-of-order packets received, allowing the sender to skip retransmission of packets it knows have been received. Finally, we have implemented forward acknowledgments [46], which both makes responses to packet loss faster and adds a mechanism to more accurately assess the state of sent packets.

The first part of the forward acknowledgment algorithm adds a retransmission trigger if the socket receives a selective acknowledgment of four packets past the last in-order acknowledgment it received. The second part notes the next sequence number to be assigned to a newly sent packet when retransmitting a packet; then, if the socket receives a selective acknowledgment for the noted sequence number before receiving one for the retransmitted packet, it considers the retransmitted packet lost and sends it once again.

3.1.2 Congestion Control. We implemented the CUBIC algorithm [24], the default congestion control algorithm used in the Linux kernel since version 2.6.19. The main variable that the algorithm controls is the congestion window (`cwnd`), which is used to determine how many packets can be sent at one time. The growth of `cwnd` changes depending on which state the congestion algorithm is in: slow start, congestion avoidance, or fast retransmit/recovery. At the beginning of the TCP connection, the algorithm starts out in slow start, where `cwnd` is incremented by one for each acknowledgment received. This leads to an exponential growth in the congestion window: For every packet sent whose sequence number fits in the growing `cwnd`, the sender will receive an acknowledgment—causing the congestion window to double after each Round-Trip Time (RTT). After the congestion control algorithm exits from slow start, either by detecting packet loss or because the congestion value exceeds the slow start threshold (`ssthresh`), the algorithm then enters congestion avoidance. During congestion avoidance, the CUBIC algorithm dictates the growth of `cwnd` by first entering a rapid concave growth followed by a convex growth which starts out slow and eventually grows exponentially. This allows the algorithm to stabilize after the "steady state" phase with concave growth, reducing the potential for congestion and packet loss, and then to enter into a "max probing" phase with convex growth in order to make sure the full capacity of the link is being utilized. When a packet loss is detected either through three duplicate acknowledgments or with the forward acknowledgment algorithm, the congestion window is reduced by



(a) Download Time, Shadow vs. NS **(b)** Congestion Window, Shadow vs. Linux

Fig. 2. The efficacy of our new TCP features in Shadow. Figure 2(a) compares Shadow to NS download times. Figure 2(b) compares congestion window over time when Shadow and Linux have the same link properties.

a multiplicative factor and then restarts in the congestion avoidance state with concave growth. A lost packet that is detected via the retransmit timer event indicates high levels of congestion. Accordingly, the congestion window is set to 1, congestion control restarts in slow start, and the retransmit timer is doubled.

3.1.3 Validation. 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. We compare its behavior to NS, a network simulator that is commonly used for performance comparisons in the networking community.[6] Although NS does not directly execute Tor (and so we are unable to use it for Tor experiments in this article), we compare Shadow to NS because NS is often validated to reality [10, 49] and NS allows us to easily model a range of 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 2(a) shows that the average download time in Shadow correlates 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 2(a).

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 100Mbit/s link between two physical Ubuntu 12.04 machines running the 3.2.0 Linux kernel. We record the `cwnd` and `ssthresh` 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 2(b) 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, Shadow is able to reproduce the overall saw-tooth pattern produced by Linux.

The results of the two experiments just discussed 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.

[6]The network simulator NS is available at http://www.isi.edu/nsnam/ns/.

3.2 Internet Topology

In addition to accurate network protocols in Shadow's simulated kernel, an accurate and realistic Internet topology model will also improve confidence in our experimental results. To ensure that we are causing the most realistic performance and congestion effects possible during simulation, we enhance Shadow's Internet representation as follows.

First, we added support in Shadow for arbitrary network graphs. Previously, the Internet topology in Shadow was represented as a complete graph with a single edge between each vertex. This model was simple and efficient, but was unable to accurately represent the complexities of autonomous systems in general and inter-domain routing in particular. A network graph representation allows us to run standard graph algorithms to compute accurate network properties, such as latency and packet loss rates, between any two vertices. We modified Shadow to use the igraph library[7] to manage the network graph and topology.

Second, we built a new network graph for Shadow in order to represent the Internet at a much finer granularity. Shadow's previous model represented entire countries as vertices and used data collected from custom PlanetLab [58] experiments, which led to missing data in places where no PlanetLab nodes existed. Using techniques from recent research in modeling Tor topologies [41, 64, 67], traceroute data from CAIDA,[8] and client/server data from the Tor Metrics Portal and Alexa,[9] we created a more realistic Internet map that includes 699,029 vertices and 1,338,590 edges. Each vertex represents either a *point of presence* or a *point of interest*:

- a *point of presence* is a cluster of all known routers with the same parent AS where the latency between each pair of routers in the cluster is within 2 milliseconds, and
- a *point of interest* is a network location of a known Internet server, Tor relay, or client.

We instrumented Shadow to assign nodes to the points of interest in this network graph, and we ran Dijkstra's shortest path algorithm to approximate routing between the nodes. Although Internet routing does not strictly adhere to shortest paths, existing work has shown that shortest path routing serves as a useful and accurate approximation of the Internet's routing behavior [18]. Previous work has also validated that routing models of the Tor network that are based on shortest path routing adhere to the *valley-free property* [64], a well-understood criteria for modeling Internet routes [19].

3.3 Tor Network Models

Shadow directly executes Tor as virtual processes that are connected through a simulated network. Although a Tor process will attempt to connect to the live, public Tor network by default, we utilize existing Tor configuration options to create a private Tor network with our own relays, clients, and servers—all hosted within the Shadow framework and without direct Internet access.

Using data from the Tor Metrics Portal [44], we configure private Shadow-Tor networks following Tor modeling best practices [31]. The experiments in the remainder of this article utilize the following network configurations, which are the largest working private experimental Tor networks to the best of our knowledge.

3.3.1 Client Behaviors. The clients in our models provide background traffic and load on the network. The *bursty* clients in our models download 320 KiB files and then wait for a time chosen

[7] igraph is a network analysis library (http://igraph.org/).
[8] The CAIDA UCSD Inferred AS Relationships Dataset - 2013 (http://www.caida.org/data/as-relationships/).
[9] We use Tor Metrics data (https://metrics.torproject.org/), and Alexa site rankings (https://www.alexa.com/).

uniformly at random from the range [1, 60,000] milliseconds after each completed download. The *bulk* clients in our models repeatedly download 5 MiB files with no pauses between completing a download and starting the next. The *TorPerf* clients download a 50 KiB, 1 MiB, or 5 MiB file over a new circuit and pause for 60 seconds after each successful download. The *TorPerf* behavior mimics the download pattern that is used in the public Tor network to benchmark performance over time [44] and allows us to understand the fidelity to the public Tor network.

3.3.2 Shadow-Tor 201307 Model. The Shadow-Tor 201307 model reflects the public Tor network as it existed in July 2013, using the then-latest stable Tor version `0.2.3.25`. The Shadow-Tor network configuration included 10 directory authorities, 3,600 relays, 13,800 clients, and 4,000 file servers. Of the clients in the model, 10,800 are *bursty* clients, 1,200 are *bulk* clients, and 1,800 are *TorPerf* clients. The ratio of these client behaviors was chosen according to known measurements of client traffic on Tor [11, 48]. Finally, in the experiments with this model we used a fixed constant packet loss rate of 0.05% between all pairs of virtual hosts.

3.3.3 Shadow-Tor 201701 Model. The Shadow-Tor 201701 model reflects the public Tor network as it existed in January 2017, using the stable Tor version `0.2.8.10`. The Shadow-Tor network configuration included a total of 2,000 Tor relays, 49,800 Tor clients, and 5,000 file servers. Of the clients in the model, 48,005 are *bursty* clients, 1,495 are *bulk* clients, and 300 are *TorPerf* clients. The ratio of client behaviors was chosen according to known measurements of Tor client traffic [34].

We use the Shadow-Tor 201701 model to explore the effects of traffic load and packet loss, KIST parameter settings, and incremental deployments. In the experiments with this model, the packet loss between each pair of virtual hosts corresponds to the following linear function of the latency between the pair (in milliseconds): $\text{packetloss} \leftarrow \min(300, \text{latency})/(300)(1.5\%)$. Accordingly, packet loss rates between hosts may range from a minimum of 0.005% to a maximum of 1.5%.

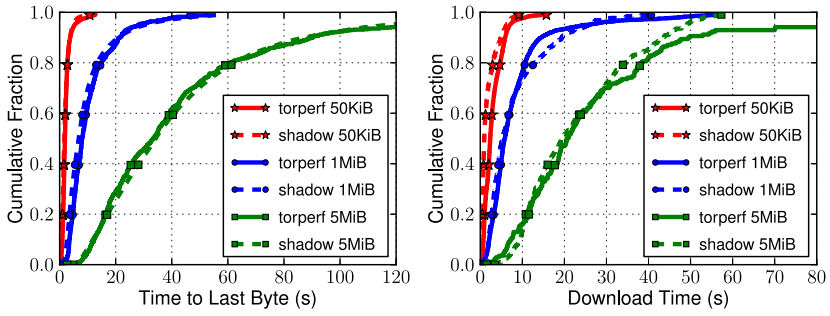
3.3.4 Discussion. Some significant changes were made between the 201307 and 201701 models. First, we changed the packet loss model because the fixed packet loss rate used in the 201307 model is not accurate in some situations; for example, we should expect more packet loss between low-bandwidth relays in the United States and Australia than we should expect between two high-bandwidth relays that are connected in the same data center. Packet loss in the 201701 model is significantly higher than in the 201307 model, so we ran more clients in the 201701 model to ensure that the relays in our Shadow-Tor network remained properly loaded. We note that measuring and modeling packet loss in Tor is a direction for future work.

Second, we changed the client type ratio between the 201307 and 201701 models. The most recently available measurements as of 2013 indicated that 58% of bytes were non-web (primarily bulk) [11], while the most recently available measurements as of 2017 indicated that 24% of bytes were non-web [34]. We updated the percentage of bulk clients from 9% in the 201307 model to 3% in the 201701 model in order to better track these changes. As a result, the percentage of total bytes transferred by bulk clients changed from 56% in the 201307 model to 29% in the 201701 model.

3.3.5 Validation. Figure 3 shows a comparison of publicly available TorPerf measurements collected on the live Tor network [44] during the model period to those collected in our private Shadow-Tor networks. As shown in Figure 3, our Shadow-Tor networks are capable of accurately representing Tor download times for all file sizes. These results give us confidence that our Shadow-Tor networks are 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



(a) Shadow-Tor vs. Tor, 201307 Model **(b)** Shadow-Tor vs. Tor, 201701 Model

Fig. 3. Comparison of live Tor measurements collected from Tor Metrics [44] to our Shadow-Tor models from 201307 in Figure 3(a) and 201701 in Figure 3(b).

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 spend 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 measure the time spent in these locations.

4.1 Congestion in the Public 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 earlier, 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.

4.1.1 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 is sent to the kernel, we log another message containing the time and ID. The difference between these times represents Tor application congestion.

4.1.2 Kernel Congestion. Measuring kernel queuing times is 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, multi-threaded library called `libkqtime`.[10]

In order to measure kernel congestion, `libkqtime` must determine when data cross the host–network boundary and when they cross the application–kernel boundary. For the host–network boundary, we rely on packet capture via `libpcap`.[11] To receive inbound and outbound packets from `libpcap`: we initialize the library and set the direction to `PCAP_D_IN` and `PCAP_D_OUT`,

[10]We have released `libkqtime` as open source software (https://github.com/robgjansen/libkqtime.git).
[11]libpcap is a portable library for network traffic capture (http://www.tcpdump.org/).

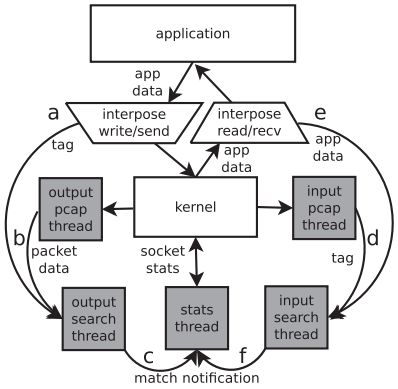


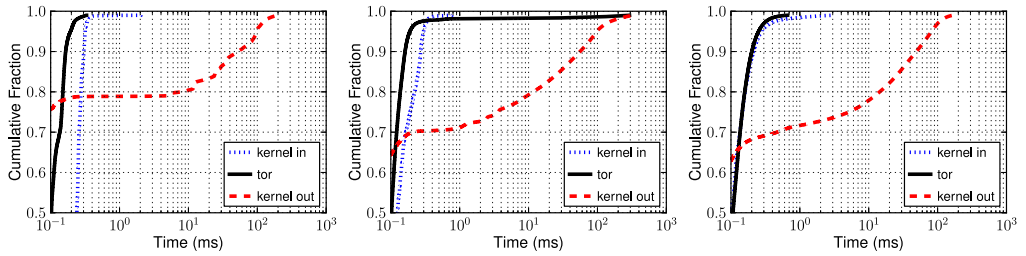
Fig. 4. An overview of the operation of `libkqtime` to measure inbound and outbound kernel delays.

respectively; we compile a filter to only capture TCP packets; and we register a callback function to handle the captured packets. For the application–kernel boundary, we use *function interposition*: We create a special Linux shared library (`libkqtime-preload.so`) containing `write()`, `send()`, `read()`, and `recv()` function signatures; and we interpose on the application's calls to these functions by preloading this library before execution by setting the environment variable "`LD_PRELOAD`" to "`libkqtime-preload.so`".

During initialization, `libkqtime` creates five helper threads that it uses to compute queuing delays. The input and output *pcap threads* register to receive inbound and outbound packets via `libpcap`, respectively. Each pcap thread asynchronously communicates with a *search thread* that performs substring matching on packet and application data. Finally, each search thread communicates with a single *stats thread* that collects socket statistics upon successful string matches. Using multiple threads in this way ensures that `libkqtime` minimally interjects in the normal operation of the application. An overview of `libkqtime` is shown in Figure 4, where the shaded shapes represent operations done in the `libkqtime` worker threads and the unshaded shapes represent actions taken in the application thread(s).

An application links to `libkqtime` and registers the socket descriptors for which it would like to gather kernel queuing delays. Then, as the application sends data to those sockets, `libkqtime` copies a 16-byte *tag* from the data and asynchronously sends it along with a timestamp to the output search thread (Figure 4(a)). We choose as a tag bytes [200,216) from the data buffer. Since Tor cells are 512 bytes, this allows us to use encrypted cell contents as a tag and avoid doing a hash and also allows us to skip Ethernet, IP, and TCP headers on incoming packets. The output pcap thread asynchronously sends outgoing packet *payloads* to the output search thread (Figure 4(b)), which itself searches the payloads for the tag. When it finds a match, it asynchronously sends the tag and match timestamps to the stats thread (Figure 4(c)). The stats thread then collects socket length and capacity information from the kernel and logs it along with the timestamps, the difference between which represents kernel congestion. The process works analogously in the inbound direction, except the tags originate from the input pcap thread (Figure 4(d)) and the payloads from the preload library (Figure 4(e)).

4.1.3 Results. To collect congestion information in Tor, we first ran three public 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 relays and used a network appliance to rate limit curiosity1 at 1Mbit/s, curiosity2 at 10Mbit/s, and curiosity3 at 50Mbit/s. Only curiosity2 had the guard flag during our data collection, signifying that it could be chosen as the



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

Fig. 5. The distribution of congestion inside Tor and the kernel on our 3 relays running in the public Tor network, as measured between Janary 20, 2014 and January 28, 2014. Most congestion occurred in the outbound kernel queues on all three relays.

entry relay for circuits. On January 20, 2014, we swapped the vanilla Tor binary with a version linked to `libkqtime` and modified as discussed in Section 4.1.1. We collected Tor and kernel congestion for 190 hours (just under eight days) ending on January 28, 2014, and then replaced the vanilla Tor binary.

The distributions of congestion as measured on each relay during the collection period are shown in Figure 5 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 more than 95% of our measurements. This finding is consistent across all three relays we measured. Kernel outbound congestion increases from curiosity1 (Figure 5(a)) to curiosity2 (Figure 5(b)), and again slightly from curiosity2 (Figure 5(b)) to curiosity3 (Figure 5(c)), indicating that the congestion that we observed is a function of relay capacity or load.

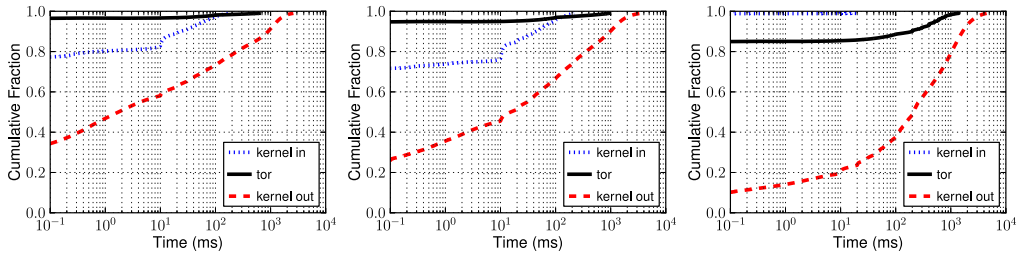
4.1.4 Ethical Considerations. We took careful precautions to ensure that our live data collection did not breach users' anonymity. In particular, *we captured only buffered data timing information*; no network addresses were ever recorded. We discussed our experimental methodology with Tor Project maintainers, who raised no objections. Finally, we contacted the Institutional Review Board (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. Note that our experiments were conducted before the establishment of the Tor Research Safety Board.[12]

4.2 Congestion in a Shadow-Tor Network

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

4.2.1 Tor and Kernel Congestion. The process for collecting congestion in Shadow is simpler than in public Tor since we have direct access to Shadow's virtual kernel. In our modified Tor, each

[12] https://research.torproject.org/safetyboard.html



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

Fig. 6. Relay congestion by circuit position in our Shadow-Tor network, measured by tracking cells from 1,200 clients (selected uniformly at random) from one end of the selected clients' circuit to the other. Most congestion occurred in the outbound kernel queues, independent of relay position.

cell again contains a unique ID as described in Section 4.1.1. The unique cell ID is used to compute congestion inside of Tor and is forwarded with the cell as it travels through the circuit, allowing us to compute end-to-end circuit congestion. However, in order for the Shadow kernel to log the time when the packet containing the cell payload is sent or received at the network interface, Tor would need to communicate the unique cell ID to the kernel for every cell. To avoid this expensive operation, we instead add a 16-byte constant magic token to each cell and include both the unique cell ID and the magic token in the payload of all cells that Tor sends. Since we prevent Tor from encrypting cell contents for efficiency reasons, the Shadow kernel can search outgoing packets for the unencrypted constant magic token immediately before they leave the virtual network interface. When the magic token is found, the Shadow kernel computes the location of the cell ID (at a known memory address offset from the magic token) and 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 Shadow timestamps, and the timestamps that are logged when a cell enters and leaves Tor, are used to compute Tor and kernel congestion.

4.2.2 Results. We use our Shadow-Tor 201307 Model from Section 3.3.2, with the addition of the cell tracking information discussed earlier. Since tracking and logging information for every cell would consume an extremely large amount of disk space, we sample congestion as follows: We select 10% 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 distribution of congestion measurements will be biased toward 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 across relays 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 6. We again find that congestion occurs most significantly in the kernel outbound queues independent of a relay's circuit position, and we also found that exit relays (Figure 6(c)) tended to have higher congestion than entry relays (Figure 6(a)) or middle relays (Figure 6(b)). Our Shadow experiments indicate higher congestion than in public Tor, which we attribute to our client-oriented sampling method described earlier.

5 TRAFFIC MANAGEMENT ANALYSIS

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 explore how this problem adversely affects Tor's traffic management by disrupting existing scheduling mechanisms to the extent that they become ineffective. Recall from Section 2.1 that each Tor relay creates and maintains a single TCP connection to every relay to which it is connected, providing a reliable and in-order data transport. All communication between two relays occurs through this single TCP connection, and, in particular, this connection multiplexes all circuits that are established between its relay endpoints.

5.1 Sequential Socket Writes

Tor uses a library called libevent[13] 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 are triggered for *one socket at a time*, regardless of the number of socket descriptors that Tor has registered and are able to be triggered. 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-Served (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.

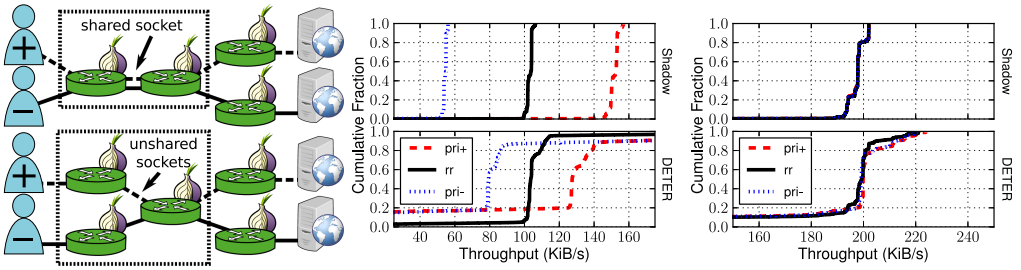
5.2 Bloated Socket Buffers

Linux uses TCP auto-tuning to dynamically and monotonically increase each socket buffer's capacity using the socket connection's bandwidth-delay product calculation [70]. 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, this is not the case for 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* 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, there is high potential for kernel queuing delays.

Tor can no longer adjust the priority of data once they are sent to the kernel, even if those data are 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-volume traffic of worse priority (typical of many bulk data transfers), while the

[13] libevent is a high-performance asynchronous event notification network library (http://libevent.org/).



(a) Socket Sharing Scenarios **(b)** Shared Socket Priority **(c)** Unshared Sockets Priority

Fig. 7. Socket sharing affects circuit priority scheduling.

other contains bursty, low-volume traffic of better priority (typical of many interactive data transfer sessions). In the absence of data on the low-volume circuit, the high-volume 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 high-volume data previously received by the kernel is flushed to the network. This problem theoretically worsens as the number of sockets increases, suggesting that research proposing that Tor use multiple sockets between each pair of relays [7, 21, 23] may be misguided.

5.3 Effects on Circuit Priority

To study the effects on circuit priority, we modified 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 modified the built-in EWMA circuit scheduler that prioritizes bursty traffic over bulk traffic [65] to include a priority factor $\mathcal{F}$: the circuit scheduler counts $\mathcal{F}$ cells for every cell scheduled on a worse-priority circuit. Therefore, the EWMA of the worse-priority class will effectively increase $\mathcal{F}$ times faster than normal, giving a scheduling advantage to better-priority traffic.

We experiment with two separate private Tor networks: one using Shadow [32] and the other using DETER.[14] We consider two clients downloading from two file servers through Tor in the scenarios shown in Figure 7(a):

- *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/s 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 on 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

[14]DETER is a general-purpose experimentation testbed that can run Tor on bare-metal hardware (http://www.isi.edu/deter).

through the circuits continuously for 10 minutes in the Shadow experiments and for 60 minutes in the DETER experiments.[15]

The results collected during each of the scenarios are shown in Figures 7(b) and 7(c). 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 round-robin scheduler ("rr"). The round-robin scheduler provides no priority and is used to demonstrate those cases where the priority scheduler also does not provide priority. As shown in Figure 7(b), performance differentiation occurs correctly with the priority scheduler on a shared socket. However, as shown in Figure 7(c), the priority scheduler is unable to differentiate circuits based on throughput when the circuits do not share a socket.

Note that the absolute difference in throughput between the shared (Figure 7(b), 100 KiB/s) and unshared (Figure 7(c), 200 KiB/s) scenarios is an artifact of our artificial limit of middle relay TCP buffer sizes to 32 KiB and should not be interpreted as an evaluation metric. The middle relay in the unshared scenario provided twice the bandwidth as the middle relay in the shared scenario because the relay in the unshared scenario had two sockets each with 32 KiB buffers.

5.4 Discussion

As outlined earlier, 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,[16] 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% of the time, clearly indicating that adjusting Tor's socket management could significantly affect data priority inside of Tor.

Note that the socket mismanagement problem is not solved simply by parallelizing the libevent notification system and the 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, we believe that they would still result in bloated buffers containing data that cannot be sent due to closed TCP congestion windows.

6 KERNEL-INFORMED SOCKET TRANSPORT

In this section we present a novel transport algorithm for Tor, called Kernel-Informed Socket Transport (KIST), that was designed to overcome the previously discussed inefficiencies resulting from Tor's socket management. KIST chooses between all circuits that have queued data and belong to any writable socket, and it dynamically adjusts the amount written to each socket based on real-time kernel information. This is done in two steps: First, it collects a set of write-pending sockets over time; and second, it runs a new socket scheduling algorithm which dynamically computes the amount of data to write to each of the write-pending sockets.

6.1 Collecting Write-Pending Sockets

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

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

[16]TorPS (Tor Path Simulator) quickly simulates path selection in Tor (http://torps.github.io/).

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 `SocketCollectionPeriod` parameter. This allows us to increase the number of candidate circuits that we consider when scheduling and writing cells to the kernel sockets: We may select among all circuits which contain cells that are waiting to be written to one of the sockets in our writable set.

ALGORITHM 1: `NotifySocketWritable()`: This function is invoked by libevent for each writable socket as they become writable over time.

Require: $sock, \mathcal{T} \leftarrow SocketCollectionPeriod$
1: $L_p \leftarrow getPendingSocketList()$
2: **if** L_p **is** $Null$ **then**
3: $\quad L_p \leftarrow newList()$
4: $\quad setPendingSocketList(L_p)$
5: $\quad createCallback(\mathcal{T}, \texttt{RunSocketScheduler()})$
6: **end if**
7: **if not** $L_p.contains(sock)$ **then**
8: $\quad L_p.add(sock)$
9: **end if**
10: $disableNotify(sock.desc, sock)$

The socket collection approach is outlined in Algorithm 1. The socket descriptor `sock` is supplied by libevent. Note that we disable notification events for the socket (in Algorithm 1, Line 10) in order to eliminate duplicate notification events during the socket collection interval.

6.2 Scheduling and Writing to Sockets

Line 5 in Algorithm 1 will cause the `RunSocketScheduler()` function to be invoked after the configured time period `SocketCollectionPeriod` has elapsed. The main jobs of the socket scheduler are to determine the order in which sockets are written and to write cells from circuit queues to the kernel. The scheduling process requires the list of write-pending sockets that were collected by Algorithm 1, L_p, and a writing policy, *wpol*.

There are three major phases to the scheduling process, which is outlined in Algorithm 2. In the first phase on Line 1, the scheduler sets write limits for each socket according to the configured *writing policy* (we describe KIST writing policies in the remainder of this subsection). In the second phase on Lines 2–10, the scheduler writes any leftover bytes that remain from previous scheduling rounds (previously flushed cells that were not completely written because the kernel socket buffer was full). Note that the sockets should be enumerated in an order that respects the order in which cells were flushed in the previous round. In the final phase, on Lines 11–20, the scheduler continues writing cells to the corresponding Tor output buffers (Line 14) and then to the socket output buffers (Line 15) of pending sockets in circuit priority order until each socket either has no remaining cells or reaches its computed write limit (Line 16).

The scheduling approach just outlined allows Tor to improve circuit priority by considering all circuits from all write-pending sockets (Algorithm 2, Line 12). We also allow the configuration of writing policies in order to provide control over how much is written to each socket. We next discuss the writing policies that we designed as part of KIST.

6.2.1 The KIST-Lite Writing Policy. As shown in Algorithm 3, in the KIST-Lite writing policy we set the write limit for each socket to *SIZE_MAX*, which effectively means the policy does not enforce artificial write limits on the socket (it is still subject to the usual kernel limits). This policy will allow us to use simulation to understand the benefits of the KIST socket scheduler without write limits. Additionally, this policy is useful for deployment contexts where the operating system on

ALGORITHM 2: `RunSocketScheduler()`: This function is invoked after `SocketCollectionPeriod` time has elapsed (see Algorithm 1, Line 5).

Require: L_p, *wpol* {configured writing policy}
1: $wpol.setWriteLimits(L_p)$
2: **for all** *sock* **in** L_p **do**
3: **if** $sock.hasBytesForKernel()$ **then**
4: $sock.writeBytesToKernel()$
5: **end if**
6: **if not** $sock.hasCells()$ **or not** $sock.canWrite()$ **then**
7: $L_p.remove(sock)$
8: $enableNotify(sock.desc, sock)$
9: **end if**
10: **end for**
11: **while not** $L_p.isEmpty()$ **do**
12: $circ \leftarrow getBestPriorityCircuitFromSockets(L_p)$
13: $sock \leftarrow circ.getSocket()$
14: $sock.flushOneCell(circ)$ {write cell to Tor outbuf}
15: $sock.writeBytesToKernel()$ {write Tor outbuf to socket outbuf}
16: **if not** $sock.hasCells()$ **or not** $sock.canWrite()$ **then**
17: $L_p.remove(sock)$
18: $enableNotify(sock.desc, sock)$
19: **end if**
20: **end while**

ALGORITHM 3: The KIST-Lite Writing Policy

Require: L_p
1: **for all** *sock* **in** L_p **do**
2: $sock.setWriteLimit(SIZE\_MAX)$
3: **end for**

which Tor is running may not support the somewhat more complex KIST writing policy that uses real-time TCP information for each socket in order to determine appropriate socket write limits.

6.2.2 The KIST Writing Policy. 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.

We show pseudocode for computing socket write limits in Algorithm 4. We first make three system calls in Lines 2–4 for each socket (i.e., TCP connection): `getsockopt` on level `SOL_SOCKET` for option `SO_SNDBUF` to get *bufcap*, the capacity of the send buffer (Line 2); `ioctl` with command `SIOCOUTQ` to get *buflen*, the current length of the send buffer (Line 3); and `getsockopt` on level `SOL_TCP` for option `TCP_INFO` to get *tcpi*, a variety of TCP state information (Line 4).

The TCP information used by KIST includes the TCP connection's maximum segment size *mss*, the congestion window *cwnd*, the number of unacked packets *una* for which the kernel is waiting for an acknowledgment from the TCP peer, and the number of *notsent* bytes that were written

ALGORITHM 4: The KIST Writing Policy

Require: L_p
1: **for all** $sock$ **in** L_p **do**
2: $bufcap \leftarrow getsockopt(sock.desc, SOL\_SOCKET, SO\_SNDBUF)$
3: $buflen \leftarrow ioctl(sock.desc, SIOCOUTQ)$
4: $tcpi \leftarrow ioctl(sock.desc, SOL\_TCP, TCP\_INFO)$
5: $bufspace \leftarrow bufcap - buflen$
6: $tcpspace \leftarrow tcpi.mss \cdot (\epsilon \cdot tcpi.cwnd - tcpi.una) - \delta \cdot tcpi.notsent$
7: $sock.setWriteLimit(\min(bufspace, tcpspace))$
8: **end for**

to the socket but not yet sent to the network.[17] The number of bytes that can be written to the buffer, *bufspace*, is computed in Line 5, and the number of bytes that TCP would allow to be sent, *tcpspace*, is computed in Line 6. The tunable parameter ϵ in Line 6 is used to ensure that the kernel can immediately send packets in response to incoming acks rather than waiting for Tor to write more data the next time that the scheduler is run. Higher values of ϵ would result in more kernel congestion but also reduce the risk of underutilizing bandwidth. The parameter δ in Line 6 can be used to tune the impact on the write limit of previously written but not yet sent bytes. We envision that δ is a binary value, where 0 means we ignore bytes that were already written to the kernel but not yet sent in a packet and 1 means we subtract them from the socket write limit. Finally, the socket write limit is set in Line 7 as the minimum of *bufspace* and *tcpspace*.

The key insights of Algorithm 4 are that TCP will not allow the kernel to send more packets than allowed by the congestion window and that the unacknowledged packets prevent the congestion window from increasing. By respecting this write limit for each socket, KIST reduces kernel queuing delays by ensuring that the data sent to the kernel are immediately sendable. The socket write limits are enforced whenever bytes are written to the kernel (Lines 4 and 15 of Algorithm 2).

6.2.3 Global Write Limit. If all sockets are sending data in parallel, it is still possible to overwhelm the kernel with more data than it can physically send to the network even when per-socket write limits are in place. Therefore, KIST also enforces a global write limit independent of the per-socket write limits. We define the global write limit for each relay as the maximum upstream bandwidth rate of the relay's host machine following our intuition that the kernel will be unable to send more data over time than the network interface is physically capable of sending. In practice, a global write limit could be calculated in a testing phase during which writes are not limited, configured manually, or estimated using other techniques such as packet trains [30].

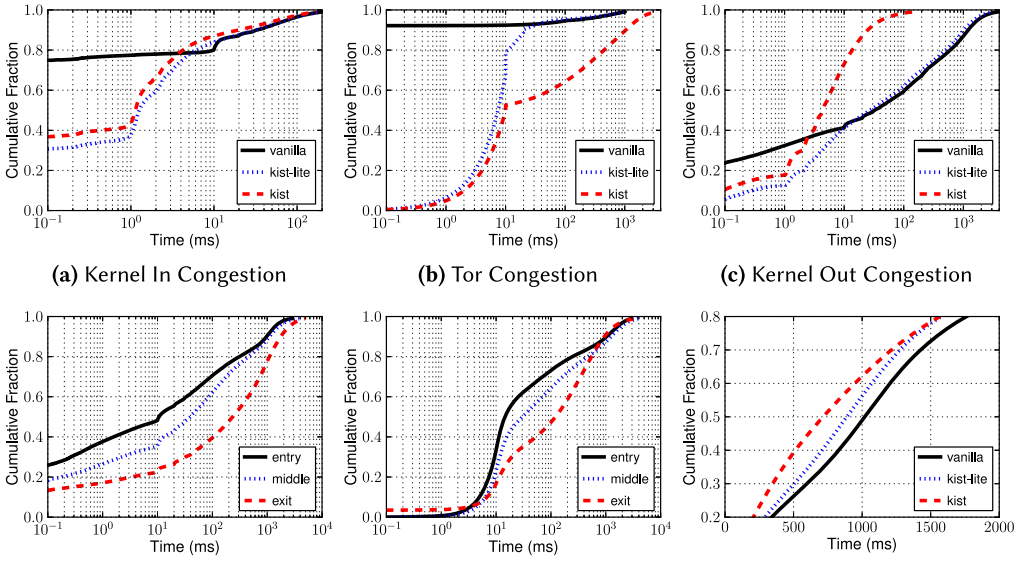
7 PERFORMANCE ANALYSIS

In this section, we use Shadow and the Shadow-Tor network models described in Section 3 to evaluate KIST and measure its effect on client performance, relay congestion, and network throughput. We further explore KIST performance and overhead when running it on a relay in the public Tor network in Section 8.

7.1 Performance of KIST Writing Policies

In this section, we compare the performance of vanilla Tor to KIST using the writing policies described in Section 6.2 and the Shadow-Tor Model 201307 described in Section 3.3.2.

[17] The Linux kernel provides *notsent* when `TCP_INFO` is queried as of version 4.6 (released May 15, 2016); on older kernels, it can be retrieved using `ioctl(2)` with request `SIOCOUTQNSD`.



(a) Kernel In Congestion **(b)** Tor Congestion **(c)** Kernel Out Congestion

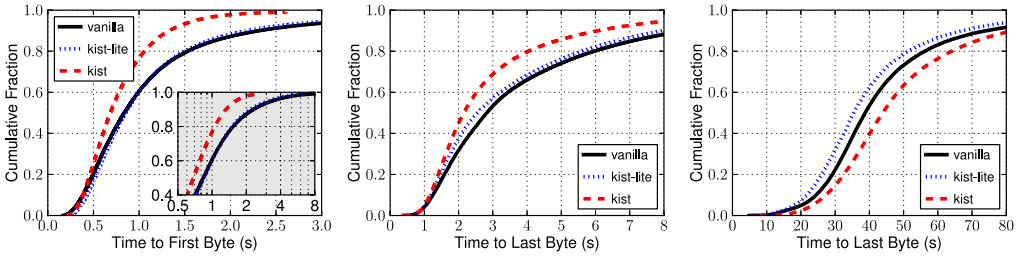
(d) Relay Congestion, Vanilla **(e)** Relay Congestion, KIST **(f)** Circuit Congestion

Fig. 8. Congestion for vanilla Tor, KIST-lite, and KIST. Figures 8(a)-8(e) show the distribution of cell congestion local to each relay (with logarithmic x-axes), while Figure 8(f) shows the distribution of the end-to-end circuit congestion for all measured cells. *Relay congestion* is the difference between the time a relay receives a cell from the network (or a downstream cell is created in case of exit) and the time it is sent back to the network (forwarded to the next-hop node). *Circuit congestion* is the sum of the relay congestion for all relays in a circuit.

7.1.1 Experiment Setup. 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, KIST with the KIST-Lite writing policy (Algorithm 3), and KIST with the KIST writing policy (Algorithm 4) using parameters $\epsilon = 1$ (we set the write limit to one congestion window worth of bytes minus the unacked data that are already in transit) and $\delta = 0$ (we ignore bytes that were previously written to the kernel but not yet sent to the network when computing the write limit). We configured a 10 millisecond `SocketCollectionPeriod` in both the KIST-Lite and KIST experiments (see Section 7.3). Note that our prototype ignores the connection enumeration order on Line 2 of Algorithm 2, an optimization that may further improve Tor's control over priority in cases where the global write limit is reached before the algorithm reaches Line 11.

7.1.2 Congestion. Recall that the goal of KIST is to move congestion from the kernel outbound queue to Tor where it can be better managed. Figure 8 shows KIST's effectiveness in this regard. In particular, Figure 8(c) shows that KIST reduces kernel outbound congestion over vanilla Tor by one to two orders of magnitude for more than 40% of the sampled cells. Further, it shows that the queue time is less than 200 milliseconds for 99% of the cells measured, compared to more than 4,000 milliseconds (4 seconds) for both vanilla Tor and KIST-Lite.

Figures 8(a) and 8(b) show how both KIST-Lite and KIST increase the congestion inside of the kernel inbound queues and in Tor. Both KIST-Lite and KIST result in sharp queue time increases up to 10 milliseconds, after which the existing 10 millisecond `SocketCollectionPeriod` timer event will fire, and Tor will flush more data to the kernel. With KIST-Lite, most of the data queued in Tor quickly get transferred to the kernel following this timeout, whereas data are queued inside of



(a) All Clients **(b)** 320 KiB *bursty* clients **(c)** 5 MiB *bulk* clients

Fig. 9. Client performance for vanilla Tor, KIST-Lite, and KIST. Figure 9(a) 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 9(b) and 9(c) show the distribution of time to complete a 320 KiB and 5 MiB file by the *bursty* and *bulk* clients, respectively.

Tor much longer when using KIST. This result is an intended feature of KIST: Tor will have more control over data priority when scheduling circuits.

We also analyze the effect of relay position on congestion. We define *relay congestion* as the difference between the time a relay receives a cell from the network (or a downstream cell is created in case of the exit) and the time the cell is sent back to the network (i.e., forwarded to the next-hop node). Relay congestion includes all of the time that a cell is queued (in kernel input buffers, Tor buffers, and kernel output buffers) or processed, but does not include any time the cell spends in transit through the network. Relay congestion is shown in Figure 8(d) for vanilla Tor and Figure 8(e) for KIST. Our experiments indicate that, for both vanilla Tor and KIST, exit relays have the most node congestion while entries have the least. We also find that KIST increases congestion by a few milliseconds for a small number of cells and by up to 10 milliseconds for about 20% of the cells, again due to our configured 10 millisecond `SocketCollectionPeriod`.

Finally, we analyze the aggregate effect of congestion on each cell. We define *circuit congestion* for each cell as the sum of the cell's relay congestion values across all relays in the cell's circuit. We note that, like relay congestion, circuit congestion excludes network transit times. The distribution of circuit congestion across all sampled cells is shown in Figure 8(f). We find that KIST reduces aggregate circuit congestion from 1010.1 milliseconds to 704.5 milliseconds in the median, a 30.3% improvement, while KIST-Lite reduces congestion by 13% to 878.8 milliseconds.

7.1.3 Performance. We show in Figure 9 how KIST affects client performance. Figure 9(a) indicates how circuit latency is generally affected by showing the time until the first byte of every download by all clients. While KIST-Lite is roughly indistinguishable from vanilla Tor, KIST reduces latency to the first byte for over 80% of the downloads: In the median, KIST reduces latency by 18.1% from 0.838 seconds to 0.686 seconds. The inset graph has a logarithmic x-axis and shows that KIST is particularly 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 9(b) and 9(c) show the distribution of time to download each 320 KiB file for the *bursty* 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 more than 40% 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 *bursty* clients pause between downloads, their traffic is often prioritized ahead of *bulk* traffic. Our results indicate

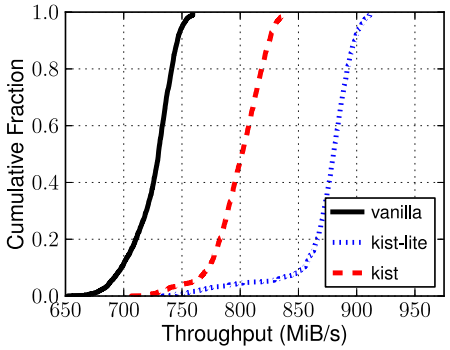


Fig. 10. Aggregate relay write throughput for the KIST-Lite and KIST writing policies increases relative to vanilla Tor.

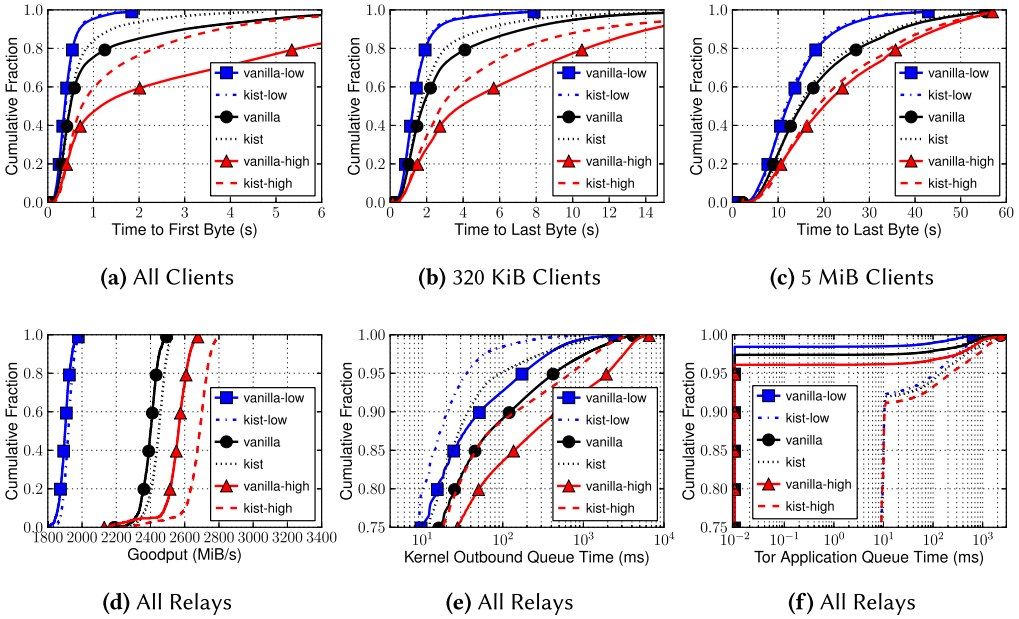
that not only does KIST decrease circuit congestion, it also increases Tor's ability to appropriately manage its traffic.

7.1.4 Throughput. We show in Figure 10 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: *Bursty* 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 KIST-Lite (a 150.1 MiB/s, or 20.5%, improvement over vanilla Tor in the median), we showed earlier that it is less effective than KIST at reducing kernel congestion and allowing Tor to correctly prioritize traffic.

7.2 Performance Under Varying Network Conditions

In this section, we compare the performance of vanilla Tor to KIST (with the KIST writing policy from Section 6.2) using the Shadow-Tor Model 201701 (Section 3.3.3) while varying traffic load and packet loss conditions in our private Shadow-Tor test network.

7.2.1 Experiment Setup. Because of the changes in Tor in the time between the construction of the 201307 and 201701 Shadow-Tor models, we felt it necessary to completely reimplement our KIST prototype for further analysis. We forked Tor at version `0.2.8.10` and refactored Tor socket scheduling code in order to allow for the implementation of multiple, distinct socket schedulers. We then implemented a new KIST scheduler (Algorithms 1 and 2) and the KIST writing policies (Algorithms 3 and 4) to change the way circuits are scheduled and to limit the amount of data that is written to the kernel. Although Line 7 of Algorithm 4 calls for a per-socket write limit of the minimum between free socket buffer space and TCP's congestion window, we found in our early experiments that the congestion window was the limiting factor in the vast majority of cases. Therefore, our prototype reduces the number of system calls per socket from three to one by ignoring socket buffer space entirely. (We expect that if the socket buffer runs out of space, the kernel will push back and propagate the socket's non-writable state to libevent, and Tor will not attempt to write to it.) Finally, we did not implement a global write limit in our prototype in order to reduce code complexity; we will show that a global write limit is unnecessary for preventing bufferbloat given that the per-socket write limits are in place. Note that the KIST prototype described earlier, including our refactoring of the socket scheduling code, has been merged into Tor version



(a) All Clients **(b)** 320 KiB Clients **(c)** 5 MiB Clients

(d) All Relays **(e)** All Relays **(f)** All Relays

Fig. 11. Client and relay performance aggregated across clients and relays for our varying traffic load models.

`0.3.2.1-alpha` on September 18, 2017, became stable in Tor version `0.3.2.9` on January 9, 2018, and is configured as the default socket scheduling algorithm.

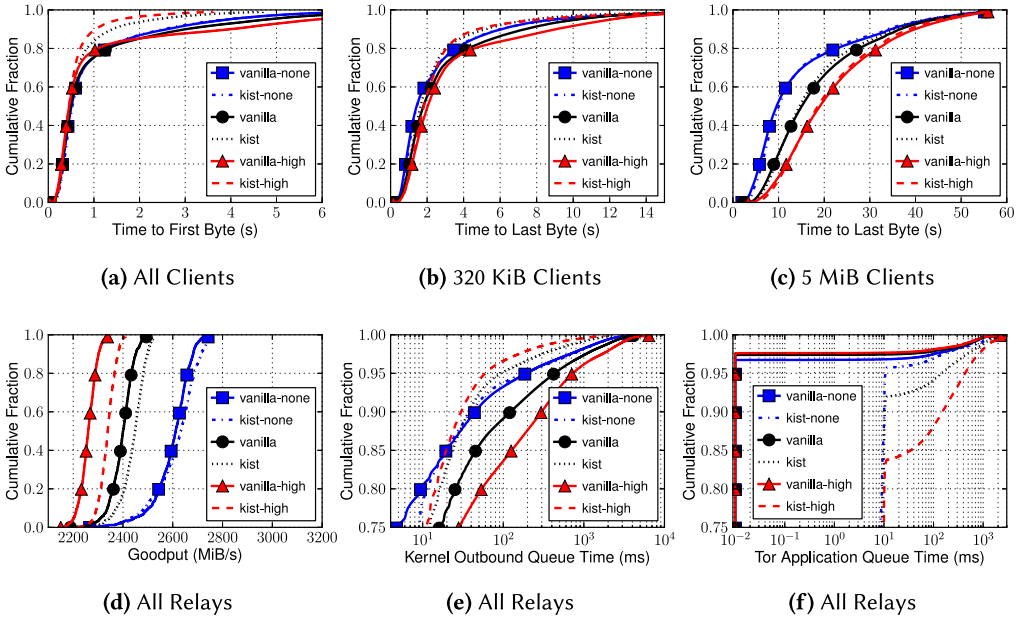
Using the prototype described earlier, we experimented with the following variants on the base Shadow-Tor model 201701 (Section 3.3.3):

- *Traffic load:* We varied the base host configuration to understand how network load affects KIST: We created *low load* and *high load* network variants by removing and adding 19,600 clients, respectively.
- *Packet loss:* We varied the base Internet model to understand how packet loss rates affect KIST: We created a *no loss* model with all packet loss rates set to 0, and we created a *high loss* model with packet loss rates double that of the base model (to a maximum of 3%).

We ran 10 Shadow experiments in total: one for vanilla Tor and one for KIST for the base Shadow-Tor model 201701 as well as each of the four variants. We set the `CircuitPriorityHalflife` config option to 30 (the default in Tor), and, in the KIST experiments, we configured a 10 millisecond `SocketCollectionPeriod` using the KIST writing policy (Algorithm 4) with parameters $\epsilon = 2$ (we set the write limit to two congestion windows worth of bytes minus the unacked data that are already in transit) and $\delta = 1$ (we also subtract the bytes that have been written to the kernel but not yet sent to the network when computing the write limit). Each experiment consumed between 0.5 and 1 TiB of RAM and simulated 45 minutes of full network activity in 6 to 8 days running on our hardware. We collected client download times, and we instrumented our relays to collect throughput information, Tor cell queuing times, and kernel queuing times.

7.2.2 Effects of Traffic Load. The performance effects of traffic load on vanilla Tor (solid lines) and KIST (dashed lines) are shown in Figure 11, where the line colors indicate the low, normal, and high load models.

Client performance is shown in Figures 11(a)-11(c) as the time to reach the first and last byte for all completed client downloads, across the low, regular, and high traffic load models. We make three



(a) All Clients **(b)** 320 KiB Clients **(c)** 5 MiB Clients

(d) All Relays **(e)** All Relays **(f)** All Relays

Fig. 12. Client and relay performance aggregated across clients and relays for our varying packet loss models.

major observations from these results. First, when there is low traffic load on the network, clients' download times are generally unaffected by the choice of socket scheduler (all of the blue lines in 11(a)-11(c) showing download times under low packet loss are roughly overlapping). Second, download times increase for both schedulers as the load on the network increases, but the increase is greater for vanilla Tor than for KIST (i.e., downloads with KIST finish more quickly than those with vanilla Tor as load increases). Third, client performance when using KIST is no worse and generally much better than when using vanilla Tor, but the improvement over vanilla Tor diminishes as download size increases (KIST improves the effectiveness of the EWMA circuit scheduler's priority mechanisms, but those benefits extend less dramatically to higher-volume flows).

Relay performance is shown in Figures 11(d)-11(f). Figure 11(d) shows that aggregate Tor network goodput per second is higher when using both KIST and vanilla Tor as the network load increases, following intuition. Goodput increases over vanilla Tor when using KIST as network load increases, but the improvement that KIST provides is most significant on the highest load model that we tested. Figure 11(e) shows that KIST generally reduces kernel queue time by more than it increases Tor queue time, as shown in Figure 11(f), suggesting that KIST is capable of reducing congestion overall rather than simply relocating it. Note that Tor queue time in Figure 11(f) is nearly zero for vanilla Tor across all three load models as Tor writes as much as possible to the kernel and tends to not queue data in the application layer. Also note that the sharp elbow at 10 milliseconds for KIST is due to the configured interval in which the socket scheduler is run.

7.2.3 Effects of Packet Loss. Figure 12 shows the performance effects of packet loss for vanilla Tor (solid lines) and KIST (dashed lines), where the line colors indicate the no packet loss, normal packet loss, and high packet loss models.

Client performance is shown in Figures 12(a)-12(c). Recall that the general trend with the varying load models was that KIST and vanilla Tor both reduced client performance as load increased, but the reduction when using KIST was less than when using vanilla Tor. However, the general trend

when varying packet loss is a bit different. When no packet loss is present, similar performance is achieved with both vanilla Tor and KIST. However, as packet loss increases, vanilla Tor tends to worsen low-volume client performance while KIST tends to improve it. Figure 12(a) and 12(b) both show that KIST actually performs best with high packet loss while vanilla Tor performs worst. Figure 12(c) shows results similar to those found for varying load models: The improvement diminishes for 5 MiB downloads since the circuit priority mechanism is operating effectively. We suspect that KIST achieves better performance for low-volume traffic at higher packet loss rates because it effectively deprioritizes high-volume circuits; the less-congested kernel can then react more quickly to low-volume traffic as Tor prioritizes its delivery to the kernel.

Relay performance is shown in Figures 12(d)-12(f). Figure 12(d) shows that aggregate Tor network goodput decreases as packet loss increases, but it decreases less when using KIST than when using vanilla Tor. Figure 12(e) shows again that KIST is able to reduce kernel queue time more as packet loss rates increase, while vanilla Tor increases kernel queue times as packet loss rates increase. Finally, the general trends in Figure 12(f) are similar to those of the Tor queue times under the varying load models: Tor queue time is nearly zero when using vanilla Tor for all tested loss models, while it is 10 milliseconds or less for more than 80% of the data when using KIST.

7.3 Performance Effects of Socket Scheduler Run Frequency

As explained in Sections 7.1.1 and 7.2.1, we configured a 10 millisecond `SocketCollectionPeriod` in our previously discussed KIST experiments. Periodically running the socket scheduler (rather than continuously) allows Tor time to queue data from multiple circuits in order to elicit data priority from the circuit scheduler. However, waiting too long between invocations of the socket scheduler could potentially cause bandwidth underutilization.

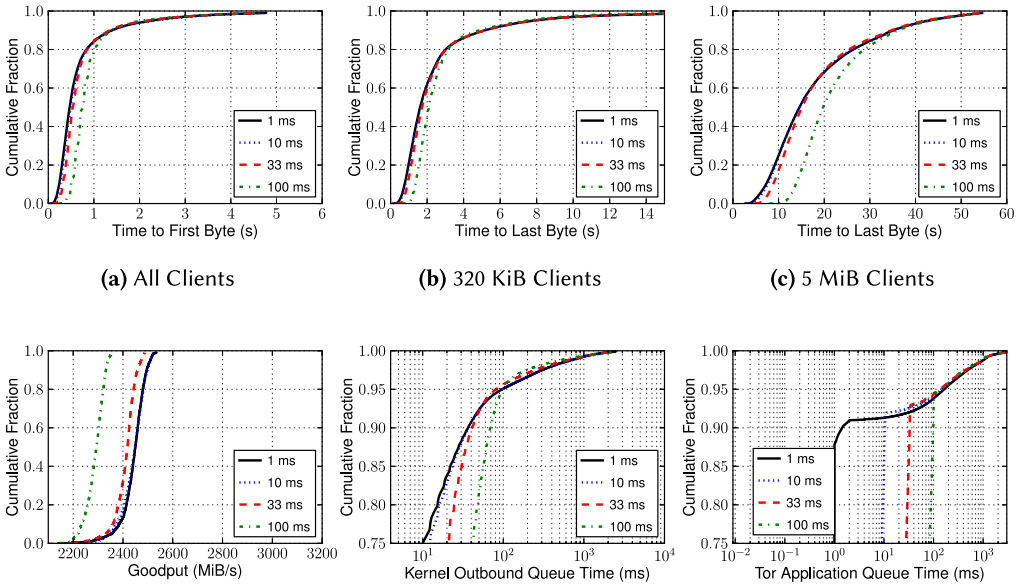
We evaluate a range of `SocketCollectionPeriod` values to better understand how the frequency with which the socket scheduler is run affects performance. In these experiments, we utilize the 201701 Shadow-Tor model and the experiment setup described in Section 7.2.1. We vary only the `SocketCollectionPeriod`, configuring it to a value between 1 and 100 milliseconds.

The performance effects of varying the `SocketCollectionPeriod` are shown in Figure 13. Figures 13(a), 13(b), and 13(c) show that client download times tend to increase for larger values of `SocketCollectionPeriod`. In the medians, the time to the last byte of 320 KiB downloads increases from 1.6 seconds for a 1 millisecond period to 2.1 seconds for a 100 millisecond period, and for 5 MiB downloads it increases from 14.6 seconds for a 1 millisecond period to 19.8 seconds for a 100 millisecond period. We do not observe a significant change in client performance between 1 and 10 millisecond periods. Relay performance also degrades for larger values of `SocketCollectionPeriod`: In the medians, relay goodput decreases from 2450 MiB/s for a 1 millisecond period to 2294 MiB/s for a 100 millisecond period (Figure 13(d)), while Tor queuing times increase according to the configured period value as expected (Figure 13(f)). Interestingly, we found that kernel outbound queuing time also increases with the `SocketCollectionPeriod`, indicating that the kernel is unable to handle the large amount of data that Tor writes after delaying the socket scheduler for longer periods.

We conclude from our results that a 10 millisecond `SocketCollectionPeriod` is reasonable because it minimally reduces performance compared to a 1 millisecond period for all but Tor queuing time, provides a long enough period over which to elicit priority, and avoids collecting TCP socket info too frequently (which results in some overhead as we describe in Section 8.3.3).

7.4 Performance in Incremental Deployments of KIST

We do not expect that all Tor relays would immediately update Tor and run KIST when it becomes available. However, KIST is incrementally deployable because it operates locally and independently



(a) All Clients **(b)** 320 KiB Clients **(c)** 5 MiB Clients

(d) All Relays **(e)** All Relays **(f)** All Relays

Fig. 13. Client and relay performance aggregated across clients and relays while varying the `Socket CollectionPeriod`, i.e., the run frequency of the KIST socket scheduler.

at each relay. Therefore, we expect that performance improvements will increase monotonically and cumulatively with the rate of deployment.

We tested our hypothesis by evaluating how performance changes across incremental deployments of KIST. We again utilize the 201701 Shadow-Tor model and the experiment setup described in Section 7.2.1. In these experiments, we configure various fractions of relays to run KIST, taking care to conduct an incremental deployment (i.e., relays that "upgrade" to KIST never "downgrade" as the deployed fraction increases).

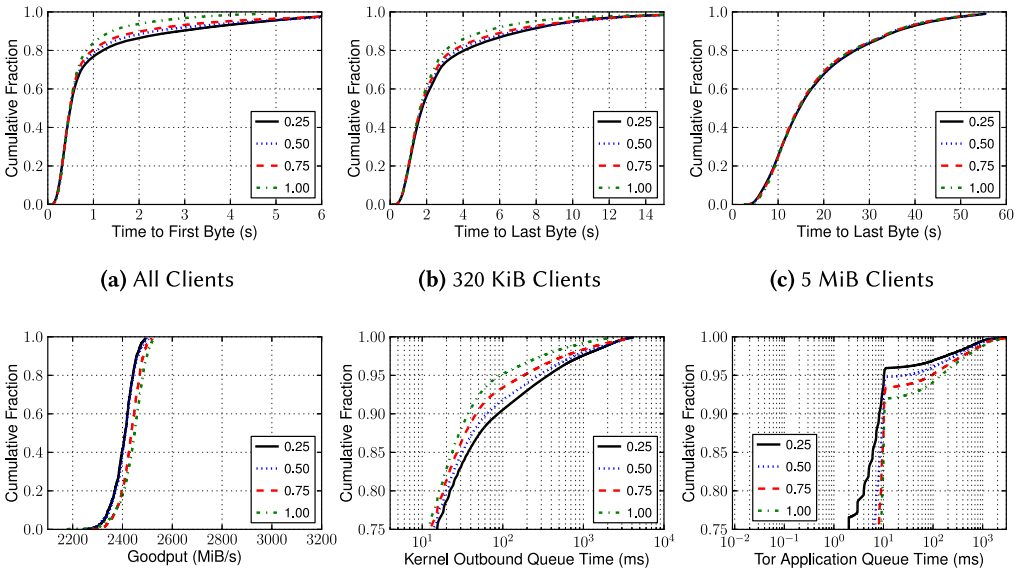
Performance as a varying fraction of relays deploy KIST is shown in Figure 14. Figures 14(a) and 14(b) show that the time to first byte and the download times for 320 KiB files generally decrease for about 40% of clients as the deployment rate increases, while Figure 14(c) shows that download times for 5 MiB files are unaffected by the deployment rate. As expected, we find that performance gains are most significant for higher priority traffic (i.e., circuits with lower EWMA throughput). We also find that relay performance follows a similar trend: As the deployment rate increases, relay goodput increases slightly (Figure 14(d)), kernel outbound queuing time decreases slightly (Figure 14(e)), and Tor application queuing time increases slightly (Figure 14(f)). We conclude from our results that Tor can benefit from incrementally deploying KIST and that any partial deployment will not decrease client or relay performance.

8 TOR NETWORK DEPLOYMENT

KIST functions independently of other relays in a circuit. We deployed KIST on a single relay under our control in the public Tor network to further understand its real-world performance effects.

8.1 Deploying KIST

We ran a Tor exit relay (with relay fingerprint `0xBCCB362660`) with the default exit policy on a bare-metal machine rented from Hurricane Electric (an Internet service provider). The machine



(a) All Clients **(b)** 320 KiB Clients **(c)** 5 MiB Clients

(d) All Relays **(e)** All Relays **(f)** All Relays

Fig. 14. Client and relay performance aggregated across clients and relays while varying the fraction of relays in our Shadow-Tor network that are deploying KIST.

had a 4-core/8-thread intel Xeon E3-1230 v5 CPU running at 3.40GHz, and was connected to an unrestricted access link capable of a symmetric bandwidth of 1Gbit/s (2Gbit/s combined transmit and receive). Several unrelated relays were co-hosted on the machine, but the average combined daily bandwidth used did not exceed 1.5Gbit/s during our evaluation period. Our deployed relay ran the prototype described in Section 7.2.1 for several weeks to allow the relay to stabilize.

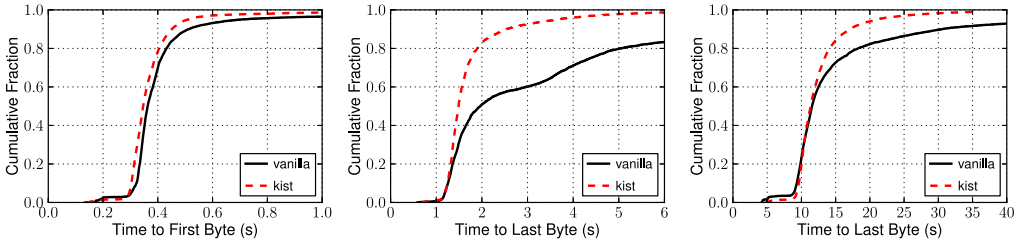
8.2 Experiment Setup

After running our relay for several weeks, we started a two-day experiment in which we ran KIST and vanilla Tor for one day each. While running KIST, our deployed relay used the KIST writing policy (Algorithm 4) using parameters $\epsilon = 2$ and $\delta = 1$ and a 10 millisecond `SocketCollectionPeriod`. We also ran three *bursty* clients that download 320 KiB files and pause for an average of 60 seconds between downloads and two *bulk* clients that download 5 MiB files and pause for 1 second between downloads. The clients choose new entry and middle relays for every circuit, but always used our deployed relay as the exit. As in the Shadow simulations in Section 7, we instrumented our relay to collect goodput, Tor cell queuing times, and kernel queuing times, and we instrumented our clients to collect file download times.

8.3 Results

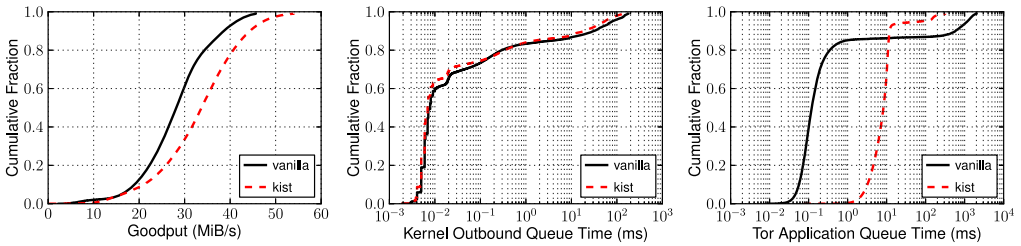
We now describe client performance, relay performance, and relay overhead results.

8.3.1 Client Performance. During our experiment, the web clients finished 7,770 downloads and the bulk clients finished 18,989 downloads. Figure 15 shows the distributions of download times recorded by our clients. While KIST reduced download times relative to vanilla Tor across all metrics, the relative improvement was greater for 320 KiB files (Figure 15(b)) than for 5 MiB files (Figure 15(c)); this could indicate that circuit priority was more effective under KIST, although we note that there may be network effects outside of our control that are also influencing these results.



(a) All Clients **(b)** 320 KiB Clients **(c)** 5 MiB Clients

Fig. 15. Client performance aggregated across five clients downloading through the live Tor network.



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

Fig. 16. Tor relay performance on our fast exit relay running in the live Tor network.

8.3.2 Relay Performance. Figure 16 shows the performance results collected on our relay. Figure 16(a) shows that KIST increased goodput relative to vanilla Tor during our observational period. Although Figure 16(b) shows an insignificant change in kernel outbound queue time, Figure 16(c) shows that KIST increased Tor application queuing time by less than 10 milliseconds (the configured socket collection period) for more than 90% of the sampled cells; we suspect that the relatively higher Tor queue time for the remaining sampled cells is due to the circuit scheduler effectively deprioritizing high-volume circuits. Additionally, KIST reduced the worst case application queue times from more than 2,000 milliseconds to less than 400 milliseconds.

8.3.3 Relay Overhead. The main overhead in KIST involves the collection of TCP information from the kernel using calls to `getsockopt`. These system calls are made for every write-pending connection after every `SocketCollectionPeriod` interval. During our experiment, we collected the time to perform the `getsockopt(2)` call to retrieve TCP information for write-pending sockets. We observed that the median number of write-pending sockets that accumulated during a 10 millisecond period was 23 (with min=1, q1=18, q3=27, and max=127), while the median amount of time to collect TCP information on all write-pending sockets was 23 microseconds (with min=1, q1=17, q3=33, and max=674). We observed a linear relationship between the amount of time required to collect TCP information on all write-pending sockets and the number of such sockets (1.08 microseconds per pending socket), independent of the total number of open sockets. Therefore, we believe that KIST overhead, with our optimization of collecting TCP information only for write-pending sockets, should be tolerable to run in the Tor main thread for even the most-utilized Tor relay.

KIST may result in a slight memory overhead. By relocating cells from the kernel address space to the Tor address space, the total amount of memory required to store cells would increase by about 2% (because Tor stores a small header for each cell that is not written to the kernel, and the

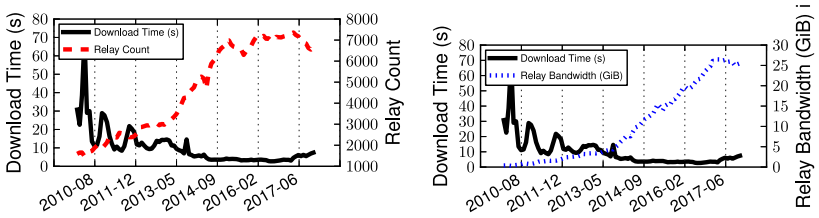


Fig. 17. Tor performance (i.e., 1 MiB file download time) correlates with relay count ($r = -.71, p < 10^{-3}$) and relay bandwidth ($r = -.58, p < 10^{-3}$).

cell payload is stored by both the kernel and Tor). Note that if the memory required by Tor increases in this regard, kernel memory pressure would be reduced by a similar amount. Furthermore, we have shown that KIST reduced end-to-end circuit congestion, which means that cells will spend less time queued overall and therefore the memory used for queuing would be used for a shorter amount of time. Finally, we note that memory usage will be bounded because Tor uses a 1,000-cell window per circuit for end-to-end circuit flow control. The maximum number of cells that would be queued at a given time on a relay for a single circuit under normal circumstances is 1,000 cells, yielding a worst-case bound of about 500 KiB per circuit.

9 SECURITY ANALYSIS

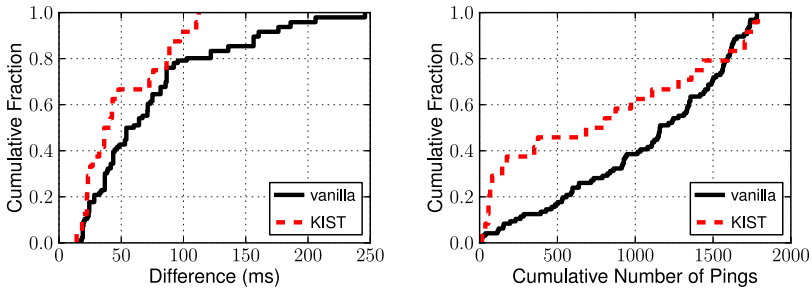
In this section, we discuss how security improves with performance (Section 9.1), show how KIST affects well-known latency-based (Section 9.2) and throughput-based (Section 9.3) performance attacks, and discuss the impact of KIST on traffic correlation attacks (Section 9.4).

9.1 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 [9, 13, 15, 35]. 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 causal factors and the difficulty of gathering useful data while preserving privacy. Whatever the causal explanation, Tor performance improvement over time (measured by the median download time of a 1 MB file) correlates with network size ($r = -.71, p < 10^{-3}$) and bandwidth metrics ($r = -.58, p < 10^{-3}$), as shown in Figure 17. (Data are from the Tor Metrics Portal [44].) Which is more relevant depends on the adversary's most significant constraints: Adversary size and distribution across the underlying network are important considerations [41].

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 previously 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 [12, 17, 20, 28, 50, 52, 54] or simply available resource observation [50], or observing other



(a) Latency Estimate Difference **(b)** Cumulative Pings until Best Estimate

Fig. 18. Latency leaks are more pronounced (18(a)) and are faster (18(b)) with KIST.

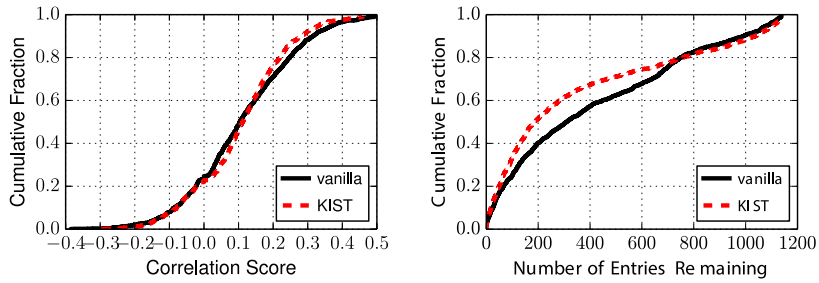
performance properties, such as latency [28]. Many papers have also explored improvements to Tor performance via scheduling, throttling, congestion management, and more (see Section 2). Manipulating performance enhancement mechanisms can turn them into potential vectors of attack themselves [38]. Geddes et al. [20] analyzed the anonymity impact of several performance enhancement mechanisms for Tor.

9.2 Latency Leak

The basic idea of a latency leak attack as set out by Hopper et al. [27] is to: (1) measure the Round-Trip Time (RTT) between a compromised exit and the client of some target connection repeatedly and then to pick the minimum result as an indication of latency; (2) compare this "min RTT" to the known, measured latency through all the hops in the circuit except the client to entry relay hop (other attacks such as throughput measurement, discussed later, are assumed to have already identified the relays in the circuit); and (3) use min RTTs and measured latencies to determine the latency between the client of the target connection and its entry relay, which is in a known network location. Such an attack 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.

We analyze the extent to which KIST affects the latency leak attack using our prototype described in Section 7.1.1 on our Shadow-Tor 201307 model from Section 3.3.2. Figure 18(a) shows the results of an experiment run 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 18(b) 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 compared to about 1,200 for vanilla Tor.

The accuracy of this latency attack significantly impacts how well an adversary can identify network location, which is what Tor is primarily designed to protect. This threat could be diminished by padding latency. More 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



(a) Throughput Correlation, Probe to Target **(b)** Entries Remaining in Candidate Set

Fig. 19. While the aggregate throughput correlations of the probe to the true entries over the set of all probes (19(a)) are not significantly affected by KIST, it is slightly easier for the adversary to eliminate candidate entries of a target client (for all clients) (19(b)) when using KIST.

significantly decrease the network location information leaked by latency. Relays can do their own RTT measurements for any edge connection and pad accordingly.

9.3 Throughput Leak

The throughput attack introduced by Mittal et al. [50] 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. This 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 volume interactive traffic and found some success, although with much less accuracy than for high-volume, bulk traffic.

We analyze the extent to which KIST affects the throughput attack, again using our prototype described in Section 7.1.1 on our Shadow-Tor 201307 model from Section 3.3.2. While measuring throughput at entry relays, we also adopt the simplification of Geddes et al. [20] 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 19(a) 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. Throughput was measured every 100ms. We found that the throughput correlations are not significantly affected by KIST.

To explain the correlation scores, recall from Section 6 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. Furthermore, 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 effect on the throughput attack, we measure how the correlation of every client's throughput to the *true* entry relay's throughput compares to the correlation of the client's throughput to that of every other *candidate* entry in the network. For every client, we start with a candidate entry set of all entries and remove those entries that have a lower correlation score with the client than the true entry's correlation score with the client. Figure 19(b) shows the distribution, over all clients, of the extent to which we were able to reduce the size of the candidate entry relay set using this heuristic. Although KIST reduced the uncertainty about the true entry 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.

9.4 Traffic Correlation

In a *traffic correlation attack*, an adversary who views some fraction of traffic entering and leaving Tor attempts to de-anonymize users by matching (i.e., correlating) the two ends of their traffic flows. The challenge in performing an attack is to accurately assess whether traffic observed between a client and its guard belongs to the same anonymous user as traffic that is observed between an exit and a destination. These attacks generally rely on detecting *patterns* in traffic flows—for example, that the quantity and spacing of data flowing on the exit↔destination link can be correlated with those of data flowing on the client↔guard link. Murdoch and Danezis demonstrated that even simple *passive* statistical approaches are sufficient to determine such correlations and de-anonymize users with high accuracy [54]. Ling et al. showed that the *active* insertion of small digital watermarks—crafted by modifying the manner in which a relay flushes and writes cells to the network—permits traffic confirmation with almost 100% accuracy and a low false-positive rate [42]. Johnson et al. quantified the live Tor network's vulnerability to passive traffic correlation attacks using a high-fidelity simulation; they showed that regular Tor users should expect to be de-anonymized by moderately provisioned adversaries with 50% probability within 3 months [41].

We reason about the effect of KIST on traffic correlation attacks using the watermarking attack introduced by Ling et al. [42] as an example. In the watermarking attack, the adversary applies a watermark by manipulating the timing of Tor cells flushed by an adversary-controlled guard or exit. This covert timing signal is then decoded by a colluding party (located at the opposite side of the Tor connection) to confirm that the traffic on the observed client↔guard and exit↔destination links belong to the same anonymous connection. Although the KIST writing policy also affects how and when Tor cells are sent to the network, an adversary deploying the watermarking attack at a guard or exit under its control could simply disable KIST to ensure that the watermark is injected into the traffic as intended. Assuming that KIST is deployed on relays in the Tor network, its use may reduce congestion and cause the adversary's timing-based watermarks to be more resilient to jitter. However, Ling et al.'s attack already approaches a 100% detection rate [42] without KIST, and so we do not expect that such attacks would be significantly improved with KIST.

Importantly, we note that KIST is not designed to protect against traffic correlation attacks. Tor is generally perceived to offer weak anonymity when an adversary can view both the ingress and egress traffic belonging to a user, and defending against such attacks without imposing large performance overheads remains an open research problem [68].

10 CONCLUSION

In this article, we outlined the results of an in-depth congestion study using both public and private Tor networks. We identified that most congestion occurs in outbound kernel buffers, and we identified multiple problems with the way in which Tor manages the process of writing to sockets.

We designed a new socket transport mechanism called KIST that chooses among *all circuits* that have queued data and belong to *any* writable socket and *dynamically adjusts the amount written* to each socket based on real-time kernel information.

We implemented multiple prototypes of KIST with the goal of transitioning it into the Tor source code and Tor network. Using these prototypes, we evaluated KIST's performance impact in simulation using multiple private Shadow-Tor networks and under a range of network load and packet loss conditions. We also deployed KIST to the public Tor network and ran it on a relay under our control for several weeks. 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 kernel queue congestion and latency while increasing relay bandwidth utilization. We also conclude that KIST is particularly effective at improving client and relay performance when a relay is under high load or high packet loss.

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.

Although our analysis is based exclusively on Linux relays (93% of Tor relays providing 92% of Tor's bandwidth run a Linux-based distribution [44]), we expect KIST to improve performance similarly across platforms because it primarily works by managing socket buffer levels. Our KIST prototype from Section 7.2.1 (including the related Tor support code) was merged on September 18, 2017 into Tor version `0.3.2.1-alpha`, in which KIST is also configured as the default socket scheduling algorithm. (KIST was marked stable in Tor version `0.3.2.9`, which was released on January 9, 2018.) In the case that operating system (OS) support for extracting the TCP information required by KIST is missing, our merged proto type gracefully falls back to running the KIST-Lite writing policy, which does not depend on special OS support.

ACKNOWLEDGMENTS

We thank Rachel Greenstadt and the anonymous reviewers of the various versions of this work for their suggestions and feedback. We thank Roger Dingledine for discussions about measuring congestion in Tor. We thank Patrick McHardy and Bryan Ford for brainstorming ideas for reducing the overhead associated with getsockopt(2). We thank Nikita Borisov, Aaron Johnson, and Prateek Mittal for discussions helpful to us for the completion and presentation of this work.

As with most Tor research, we found it useful to communicate with the Tor developers early and often. They are experts in developing and maintaining anonymous communication systems and their collaboration and feedback has greatly improved the quality of this work. We thank Andrea Shepard for implementing an initial version of Tor's socket scheduling code. We thank David Goulet, Nick Mathewson, and Tim Wilson-Brown for their assistance and feedback during the development of the production prototype and for their support in merging KIST into Tor.

REFERENCES

[1] Alessandro Acquisti, Roger Dingledine, and Paul Syverson. 2003. On the economics of anonymity. In *Financial Cryptography and Data Security (FC)*.

[2] Masoud Akhoondi, Curtis Yu, and Harsha V. Madhyastha. 2012. LASTor: A low-latency AS-aware Tor client. In *Proceedings of the IEEE Symposium on Security and Privacy (Oakland)*.

[3] M. Allman, V. Paxson, and E. Blanton. 2009. TCP Congestion Control. *RFC 5681 (Draft Standard)*. (Sept. 2009). http://www.ietf.org/rfc/rfc5681.txt.

[4] Mashael Alsabah, Kevin Bauer, Tariq Elahi, and Ian Goldberg. 2013. The path less travelled: Overcoming Tor's bottlenecks with traffic splitting. In *Proceedings of the Privacy Enhancing Technologies Symposium (PETS)*.

[5] Mashael AlSabah, Kevin Bauer, and Ian Goldberg. 2012. Enhancing Tor's performance using real-time traffic classification. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*.
[6] Mashael AlSabah, Kevin Bauer, Ian Goldberg, Dirk Grunwald, Damon McCoy, Stefan Savage, and Geoffrey Voelker. 2011. DefenestraTor: Throwing out Windows in Tor. In *Proceedings of the Privacy Enhancing Technologies Symposium (PETS)*.
[7] Mashael AlSabah and Ian Goldberg. 2013. PCTCP: Per-circuit TCP-over-IPsec transport for anonymous communication overlay networks. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*.
[8] Simurgh Aryan, Homa Aryan, and J. Alex Halderman. 2013. Internet censorship in Iran: A first look. In *Proceedings of the USENIX Workshop on Free and Open Communications on the Internet (FOCI)*.
[9] Adam Back, Ulf Möller, and Anton Stiglic. 2001. Traffic analysis attacks and trade-offs in anonymity providing systems. In *Proceedings of the Workshop on Information Hiding (IH)*.
[10] Maurizio Casoni, Carlo Augusto Grazia, Martin Klapez, and Natale Patriciello. 2015. Implementation and validation of TCP options and congestion control algorithms for Ns-3. In *Proceedings of the Workshop on Ns-3 (WNS3)*.
[11] A. Chaabane, P. Manils, and M. A. Kaafar. 2010. Digging into anonymous traffic: A deep analysis of the tor anonymizing network. In *Proceedings of the IEEE Conference on Network and System Security (NSS)*.
[12] Eric Chan-Tin, Jiyoung Shin, and Jiangmin Yu. 2013. Revisiting circuit clogging attacks on Tor. In *Proceedings of the IEEE Conference on Availability, Reliability and Security (ARES)*.
[13] Roger Dingledine and Nick Mathewson. 2006. Anonymity loves company: Usability and the network effect. In *Proceedings of the Workshop on the Economics of Information Security (WEIS)*.
[14] Roger Dingledine, Nick Mathewson, and Paul Syverson. 2004. Tor: The second-generation onion router. In *Proceedings of the USENIX Security Symposium (USENIX)*.
[15] Roger Dingledine, Nick Mathewson, and Paul Syverson. 2007. Deploying low-latency anonymity: Design challenges and social factors. *IEEE Security & Privacy* 5, 5 (Sept./Oct. 2007).
[16] Roger Dingledine and Steven J. Murdoch. 2009. *Performance Improvements on Tor or, Why Tor is Slow and What We're Going to Do About It.* Technical Report 2009-11-001. The Tor Project.
[17] Nathan S. Evans, Roger Dingledine, and Christian Grothoff. 2009. A practical congestion attack on Tor using long paths. In *Proceedings of the USENIX Security Symposium (USENIX)*.
[18] Paul Francis, Sugih Jamin, Cheng Jin, Yixin Jin, Danny Raz, Yuval Shavitt, and Lixia Zhang. 2001. IDMaps: A global internet host distance estimation service. *IEEE/ACM Transactions on Networking* 9, 5 (2001), 525–540.
[19] Lixin Gao. 2001. On inferring autonomous system relationships in the internet. *IEEE/ACM Transactions on Networking (ToN)* 9, 6 (2001), 733–745.
[20] John Geddes, Rob Jansen, and Nicholas Hopper. 2013. How low can you go: Balancing performance with anonymity in Tor. In *Proceedings of the Privacy Enhancing Technologies Symposium (PETS)*.
[21] John Geddes, Rob Jansen, and Nicholas Hopper. 2014. IMUX: Managing tor connections from two to infinity, and beyond. In *Proceedings of the ACM Workshop on Privacy in the Electronic Society (WPES)*.
[22] Mainak Ghosh, Miles Richardson, Bryan Ford, and Rob Jansen. 2014. A TorPath to TorCoin: Proof-of-bandwidth altcoins for compensating relays. In *Proceedings of theWorkshop on Hot Topics in Privacy Enhancing Technologies (HotPETs)*.
[23] Deepika Gopal and Nadia Heninger. 2012. Torchestra: Reducing interactive traffic delays over Tor. In *Proceedings of the ACM Workshop on Privacy in the Electronic Society (WPES)*.
[24] Sangtae Ha, Injong Rhee, and Lisong Xu. 2008. CUBIC: A new TCP-friendly high-speed TCP variant. *ACM SIGOPS Operating Systems Review* 42, 5 (2008), 64–74.
[25] Sebastian Hahn and Karsten Loesing. 2010. *Privacy-preserving Ways to Estimate the Number of Tor Users.* Technical Report 2010-11-001. Tor Project.
[26] Nicholas Hopper. 2013. *Protecting Tor from Botnet Abuse in the Long Term.* Technical Report 2013-11-001. The Tor Project.
[27] Nicholas Hopper, Eugene Y. Vasserman, and Eric Chan-Tin. 2007. How much anonymity does network latency leak? In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*. Expanded and revised version in [28].
[28] Nicholas Hopper, Eugene Y. Vasserman, and Eric Chan-Tin. 2010. How much anonymity does network latency leak? *ACM Transactions on Information and System Security (TISSEC)* 13, 2 (Feb. 2010), 13–28.
[29] Mohsen Imani, Armon Barton, and Matthew Wright. 2017. Forming guard sets using AS relationships. *Proceedings on Privacy Enhancing Technologies (PoPETs)* 2017, 3 (2017).
[30] Raj Jain and Shawn Routhier. 1986. Packet trains–measurements and a new model for computer network traffic. *IEEE Selected Areas in Communications* 4, 6 (1986), 986–995.
[31] Rob Jansen, Kevin Bauer, Nicholas Hopper, and Roger Dingledine. 2012. Methodically modeling the Tor network. In *Proceedings of the USENIX Workshop on Cyber Security Experimentation and Test (CSET)*.

[32] Rob Jansen and Nicholas Hopper. 2012. Shadow: Running Tor in a box for accurate and efficient experimentation. In *Proceedings of the USENIX Security Symposium (USENIX)*.
[33] Rob Jansen, Nicholas Hopper, and Yongdae Kim. 2010. Recruiting new Tor relays with BRAIDS. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*.
[34] Rob Jansen and Aaron Johnson. 2016. Safely measuring Tor. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*.
[35] Rob Jansen, Aaron Johnson, and Paul Syverson. 2013. LIRA: Lightweight incentivized routing for anonymity. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*.
[36] Rob Jansen, Marc Juarez, Rafael Galvez, Tariq Elahi, and Claudia Diaz. 2018. Inside job: Applying traffic analysis to measure Tor from within. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*.
[37] Rob Jansen, Andrew Miller, Paul Syverson, and Bryan Ford. 2014. From onions to shallots: Rewarding Tor relays with TEARS. In *Proceedings of the Workshop on Hot Topics in Privacy Enhancing Technologies (HotPETs)*.
[38] Rob Jansen, Paul Syverson, and Nicholas Hopper. 2012. Throttling Tor bandwidth parasites. In *Proceedings of the USENIX Security Symposium (USENIX)*.
[39] Aaron Johnson, Rob Jansen, Nicholas Hopper, Aaron Segal, and Paul Syverson. 2017. PeerFlow: Secure load balancing in Tor. *Proceedings on Privacy Enhancing Technologies (PoPETs)* 2017, 2 (2017).
[40] Aaron Johnson, Rob Jansen, Aaron D. Jaggard, Joan Feigenbaum, and Paul Syverson. 2017. Avoiding the man on the wire: Improving Tor's security with trust-aware path selection. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*.
[41] Aaron Johnson, Chris Wacek, Rob Jansen, Micah Sherr, and Paul Syverson. 2013. Users get routed: Traffic correlation on Tor by realistic adversaries. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*.
[42] Zhen Ling, Junzhou Luo, Wei Yu, Xinwen Fu, Dong Xuan, and Weijia Jia. 2009. A new cell counter based attack against Tor. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*. 578–589.
[43] Zhuotao Liu, Yushan Liu, Philipp Winter, Prateek Mittal, and Yih-Chun Hu. 2017. TorPolice: Towards enforcing service-defined access policies for anonymous communication in the Tor network. In *Proceedings of the International Conference on Network Protocols (ICNP)*.
[44] Karsten Loesing, Steven J. Murdoch, and Roger Dingledine. 2010. A case study on measuring statistical data in the Tor anonymity network. In *Proceedings of the Workshop on Ethics in Computer Security Research (WECSR)*. For Tor metrics statistics and data-sets, see https://metrics.torproject.org.
[45] Nick Mathewson. 2004. Evaluating SCTP for Tor. http://archives.seul.org/or/dev/Sep-2004/msg00002.html. (September 2004). Listserv posting.
[46] Matthew Mathis and Jamshid Mahdavi. 1996. Forward acknowledgement: Refining TCP congestion control. *ACM SIGCOMM Computer Communication Review* 26, 4 (1996), 281–291.
[47] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. 1996. TCP Selective Acknowledgment Options. *RFC 2018 (Proposed Standard)*. (Oct. 1996). http://www.ietf.org/rfc/rfc2018.txt.
[48] Damon McCoy, Kevin Bauer, Dirk Grunwald, Tadayoshi Kohno, and Douglas Sicker. 2008. Shining light in dark places: Understanding the Tor network. In *Proceedings of the Privacy Enhancing Technologies Symposium (PETS)*.
[49] Dharmendra Kumar Mishra, Pranav Vankar, and Mohit P. Tahiliani. 2016. TCP evaluation suite for Ns-3. In *Proceedings of the Workshop on Ns-3 (WNS3)*.
[50] Prateek Mittal, Ahmed Khurshid, Joshua Juen, Matthew Caesar, and Nikita Borisov. 2011. Stealthy traffic analysis of low-latency anonymous communication using throughput fingerprinting. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*.
[51] W. Brad Moore, Chris Wacek, and Micah Sherr. 2011. Exploring the potential benefits of expanded rate limiting in Tor: Slow and steady wins the race with Tortoise. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*.
[52] Steven J. Murdoch. 2006. Hot or not: Revealing hidden services by their clock skew. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*.
[53] Steven J. Murdoch. 2011. *Comparison of Tor Datagram Designs*. Technical Report 2011-11-001. The Tor Project.
[54] Steven J. Murdoch and George Danezis. 2005. Low-cost traffic analysis of Tor. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*.
[55] Michael F. Nowlan, Nabin Tiwari, Janardhan Iyengar, Syed Obaid Amin, and Bryan Ford. 2012. Fitting square pegs through round pipes. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.
[56] Michael F. Nowlan, David Wolinsky, and Bryan Ford. 2013. Reducing latency in Tor circuits with unordered delivery. In *Proceedings of the USENIX Workshop on Free and Open Communications on the Internet (FOCI)*.
[57] V. Paxson, M. Allman, J. Chu, and M. Sargent. 2011. Computing TCP's Retransmission Timer. *RFC 6298 (Proposed Standard)*. (June 2011). http://www.ietf.org/rfc/rfc6298.txt.

[58] Larry Peterson, Steve Muir, Timothy Roscoe, and Aaron Klingaman. 2006. *PlanetLab Architecture: An Overview*. Technical Report. PlanetLab Consortium.
[59] Joel Reardon and Ian Goldberg. 2009. Improving Tor using a TCP-over-DTLS tunnel. In *Proceedings of the USENIX Security Symposium (USENIX)*.
[60] Florentin Rochet and Olivier Pereira. 2017. Waterfilling: Balancing the Tor network with maximum diversity. *Proceedings on Privacy Enhancing Technologies (PoPETs)* 2017, 2 (2017).
[61] Micah Sherr, Andrew Mao, William R. Marczak, Wenchao Zhou, Boon Thau Loo, and Matt Blaze. 2010. A3: An extensible platform for application-aware anonymity. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*.
[62] Fatemeh Shirazi, Matthias Goehring, and Claudia Diaz. 2015. Tor experimentation tools. In *Proceedings of the International Workshop on Privacy Engineering (IWPE)*.
[63] Robin Snader and Nikita Borisov. 2008. A tune-up for Tor: Improving security and performance in the Tor network. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*.
[64] Henry Tan, Micah Sherr, and Wenchao Zhou. 2016. Data-plane Defenses against routing attacks on Tor. *Proceedings on Privacy Enhancing Technologies (PoPETs)* 2016, 4 (2016).
[65] Can Tang and Ian Goldberg. 2010. An improved algorithm for Tor circuit scheduling. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*.
[66] Florian Tschorsch and Björn Scheuermann. 2016. Mind the gap: Towards a backpressure-based transport protocol for the Tor network. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.
[67] Chris Wacek, Henry Tan, Kevin Bauer, and Micah Sherr. 2013. An empirical evaluation of relay selection in Tor. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*.
[68] Ryan Wails, Yixin Sun, Aaron Johnson, Mung Chiang, and Prateek Mittal. 2018. Tempest: Temporal dynamics in anonymity systems. *Proceedings on Privacy Enhancing Technologies (PoPETs)* 2018, 3 (2018).
[69] Tao Wang, Kevin Bauer, Clara Forero, and Ian Goldberg. 2012. Congestion-aware path selection for Tor. In *Proceedings of the Financial Cryptography and Data Security (FC)*.
[70] Eric Weigle and Wu-chun Feng. 2002. A comparison of TCP automatic tuning techniques for distributed computing. In *Proceedings of the IEEE Symposium on High Performance Distributed Computing (HPDC)*.

Received December 2017; revised June 2018; accepted September 2018