Technical Report

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

TR 14-012

Appendices to Accompany “Never Been KIST: Tors Congestion Management Blossoms with Kernel-Informed Socket Transport”

Rob Jansen, John Geddes, Chris Wacek, Micah Sherr, and Paul Syverson

June 06, 2014

Appendices to Accompany “Never Been KIST: Tor’s Congestion Management Blossoms with Kernel-Informed Socket Transport”

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

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

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

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

Abstract

This document provides appendices to accompany the publication entitled “Never Been KIST: Tor’s Congestion Management Blossoms with Kernel-Informed Socket Transport” to appear in the *Proceedings of the 23rd USENIX Security Symposium*, 2014 [8].

Appendices

A Enhancing Shadow

This section provides implementation details about the Shadow enhancements outlined in Section 3 [8].

A.1 Implementing TCP

In order to accurately simulate the real world Tor network, we needed to make sure that Shadow implements the many-faceted interconnected algorithms used in the TCP kernel stack for congestion control. These algorithms deal with two primary aspects, growth of the congestion window (which helps dictate how much data can be sent at time) and how to detect and respond to packet loss. We will briefly discuss the TCP algorithms incorporated into Shadow, in addition to looking at how Shadow compares to both the TCP implementation in Linux and to the network simulator (NS) [12], a tool very commonly used in the network community when comparing performance of different TCP algorithms.

Detecting Packet Loss: We have implemented in Shadow four techniques for detecting and handling packet loss commonly found in TCP implementations. First, we compute the retransmission timer value [13] and set up events to notify Shadow when a packet is lost and should be retransmitted. Second, as part of fast retransmit/recovery [6], once 3 duplicate acknowledgments have been received, all unacknowledged packets are considered lost and will be resent. Third, we use selective acknowledgments [11] 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 acknowledgment [10], which both makes response to packet loss faster and adds a way to more accurately assess the state of sent packets.

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

Congestion Control: For congestion control we implemented the CUBIC algorithm [7], the default congestion control algorithm used in the Linux kernel since version 2.6.19. The main variable the algorithm controls is the congestion window (`cwnd`), which is used to help TCP determine how many packets can be sent at one time. The growth of `cwnd` changes depending 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 for each acknowledgment received `cwnd` is incremented by one. This leads to an exponential growth in the congestion window, as for every `cwnd` packets sent the sender will receive `cwnd` acknowledgments every round-trip time, doubling the congestion window each RTT. After the congestion control algorithm exits from slow start, either by detecting packet loss or having the congestion value exceed 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 with flat lines for a while, then enters convex growth which starts out slow and eventually

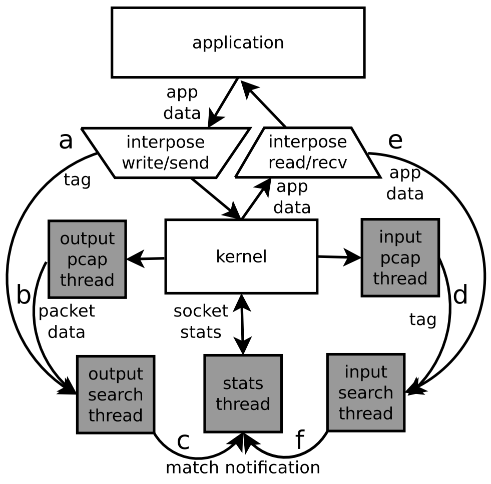


Figure 1: An overview of the operation of `libkqtime` to measure inbound and outbound kernel delays.

grows exponentially. This allows the algorithm to stabilize after the "steady state" phase with concave growth, reducing the potential for congestion and packet loss, then enter into a "max probing" with convex growth in order to make sure the full capacity of the link is being utilized. When a packet loss is detected through either three duplicate acknowledgments or with the forward acknowledgment algorithm, the congestion window is reduced by a multiplicative factor and then starts over in the congestion avoidance state with concave growth. If a packet loss is detected via the retransmit timer event, this indicates high levels of congestion. Accordingly, the congestion window is set to 1, congestion control starts over in slow start, and the retransmit timer is doubled.

A.2 New 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 Internet representation as follows.

First, we enhance Shadow's internal topology representation to support arbitrary network graphs. Previously, Shadow's topology 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 [3] library to manage the network graph and topology.

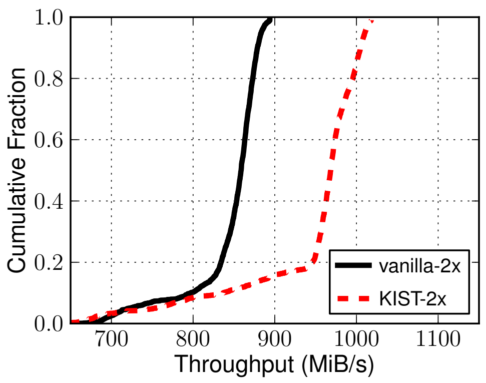
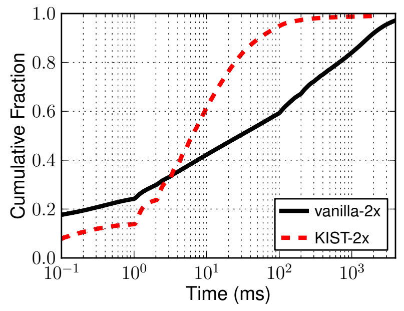


Figure 2: Aggregate relay write throughput for vanilla Tor and KIST. Shown is aggregate relay write throughput under a heavily-loaded Shadow-Tor network with twice as many clients (27,600) as discussed in our Tor model in Section 3 [8]. As when under normal load, KIST increases network throughput over vanilla Tor.

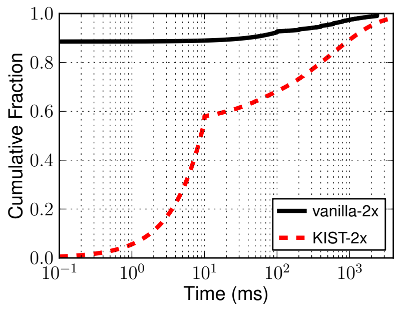
Second, we enhance the topology fed into Shadow to represent the Internet at a much finer granularity. Shadow's previous model represented entire countries as vertices and used data collected from custom PlanetLab [14] experiments, which led to missing data in places where no PlanetLab nodes exist. Using techniques from recent research in modeling Tor topologies [9,15], traceroute data from CAIDA [2], and client/server data from the Tor Metrics Portal [5] and Alexa [1], we created a much 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; a point of interest represents the 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 while running Dijkstra's shortest path algorithm to approximate routing between the nodes.

B Measuring Kernel Congestion with libkqtime

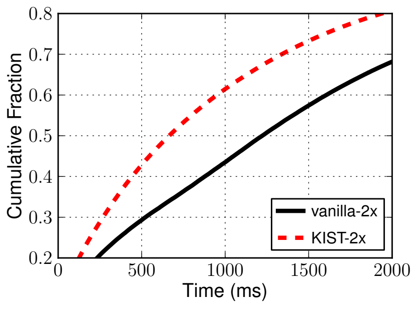
In order to measure kernel congestion, `libkqtime` must determine when data crosses the host/network boundary and when it crosses the application/kernel boundary. For the host/network boundary, we rely on packet capture via `libpcap` [4]. To receive inbound and outbound packets from `libpcap`: we initialize the library and set the direction to `PCAP_D_IN` and `PCAP_D_OUT`, respectively; we compile a filter to only capture TCP packets; and we register a callback function to handle the captured pack-



(a) Kernel Out Congestion

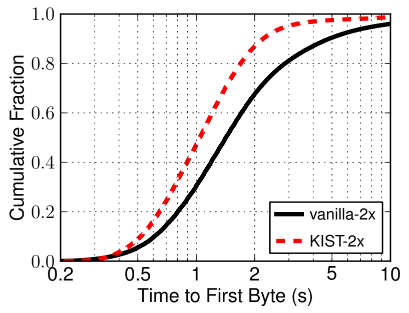


(b) Tor Congestion

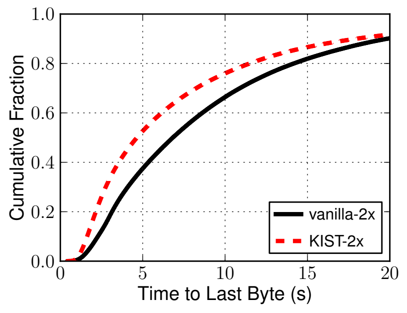


(c) Circuit Congestion

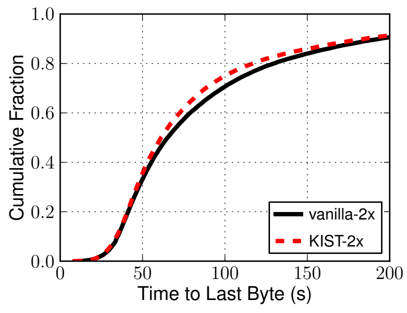
Figure 3: Congestion for vanilla Tor and KIST. Shown is performance under a heavily-loaded Shadow-Tor network with twice than many clients (27,600) as discussed in our Tor model in Section 3 [8]. Figures 3b and 3a show the distribution of cell congestion local to each relay (with logarithmic x-axes), while Figure 3c shows the distribution of the end-to-end circuit congestion for all measured cells.



(a) All Clients



(b) 320 KiB “web” clients



(c) 5 MiB “bulk” clients

Figure 4: Client performance for vanilla Tor and KIST. Shown is performance under a heavily-loaded Shadow-Tor network with twice as many clients (27,600) than discussed in our Tor model in Section 3 [8]. Figure 4a shows the distribution of the time until the client receives the first byte of the data payload, for all clients, while Figures 4b and 4c show the distribution of time to complete a 320 KiB and 5 MiB file by the “web” and “bulk” clients, respectively.

ets. 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 environmental variable `LD_PRELOAD=libkqtime-preload.so`.

During initialization, `libkqtime` creates 5 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 will perform substring matching on packet and application data. Finally, each search thread communicates with a single *stats thread* that will collect 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 1, 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 1a). The output pcap thread asynchronously sends outgoing packet *payloads* to the output search thread (Figure 1b), 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 1c). 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 1d) and the payloads from the preload library (Figure 1e).

C Heavily Loaded Network Results

To better understand the benefits provided by KIST, we tested it under a more heavily loaded network. In this set of experiments, we configured our Shadow-Tor network with twice the number of each client type described in Section 3 [8]. This resulted in 27,800 clients producing load on the exisitng 3,600 relays. All other settings remain as before. Figures 3, 4, and 2 compare congestion, performance, and aggregate relay throughput in vanilla Tor and KIST, respectively.

References

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

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

[3] igraph network analysis package. http://igraph.org/.

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

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

[6] Allman, M., Paxson, V., and Blanton, E. TCP Congestion Control. RFC 5681 (Draft Standard), Sept. 2009.

[7] Ha, S., Rhee, I., and Xu, L. CUBIC: a new TCP-friendly high-speed TCP variant. *ACM SIGOPS Operating Systems Review 42*, 5 (2008), 64–74.

[8] Jansen, R., Geddes, J., Wacek, C., Sherr, M., and Syverson, P. Never been KIST: Tor's congestion management blossoms with kernel-informed socket transport. In *USENIX Security Symposium (USENIX)* (2014).

[9] Johnson, A., Wacek, C., Jansen, R., Sherr, M., and Syverson, P. Users get routed: Traffic correlation on tor by realistic adversaries. In *ACM Conference on Computer and Communications Security (CCS)* (2013).

[10] Mathis, M., and Mahdavi, J. Forward acknowledgement: Refining TCP congestion control. *ACM SIGCOMM Computer Communication Review 26*, 4 (1996), 281–291.

[11] Mathis, M., Mahdavi, J., Floyd, S., and Romanow, A. TCP Selective Acknowledgment Options. RFC 2018 (Proposed Standard), Oct. 1996.

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

[13] Paxson, V., Allman, M., Chu, J., and Sargent, M. Computing TCP's Retransmission Timer. RFC 6298 (Proposed Standard), June 2011.

[14] Peterson, L., Muir, S., Roscoe, T., and Klingaman, A. PlanetLab Architecture: An Overview. Tech. rep., PlanetLab Consortium, 2006.

[15] Wacek, C., Tan, H., Bauer, K., and Sherr, M. An empirical evaluation of relay selection in Tor. In *Network and Distributed System Security Symposium (NDSS)* (2013).