Reducing Latency in Tor Circuits with Unordered Delivery

Michael F. Nowlan, David Wolinsky, and Bryan Ford
Yale University

Abstract

Tor, the popular anonymous relay tool, incurs significant latency costs—partly due to extra network hops, but also due to TCP's strict in-order delivery. We examine the problem of TCP's head-of-line blocking in Tor, although this problem affects any application multiplexing streams atop TCP. Using uTCP and uTLS, techniques for enabling unordered delivery in TCP and TLS, respectively, we eliminate head-of-line blocking between Tor circuits sharing a TCP connection, without sacrificing Tor's security. The small code footprint of uTCP and uTLS, and the minimal changes required to Tor, suggest the feasability of our approach. A micro-benchmark indicates that the integration of uTCP and uTLS can noticeably lower application-perceived latency.

1 Introduction

Tor [10] is a protocol and open-source tool providing increased anonymity to network communication by routing it through a dynamically-chosen set of *relay* servers. In addition to hiding the communicating endpoints' identities, Tor also obfuscates its use by "blending in" with regular encrypted traffic (e.g. e-commerce). Tor users observe significantly increased latency partly due to the extra hops imposed by the relay scheme, but also due to side effects of Tor's use of TLS [9] over TCP [27] as its general transport substrate. These side effects may manifest in any application that uses TCP in this way.

TCP offers an attractive underlying substrate for a network overlay for reasons including reliability, congestion control and widespread network support. In the case of Tor and other communication overlays [18, 28], the use of TCP goes even further to help provide *unobservability*, wherein a malicious eavesdropper cannot easily distinguish overlay traffic from regular Internet traffic [16]. Despite its benefits, TCP only supports strict in-order delivery, requiring a node to delay delivery of subsequent packets after a lost packet. In overlays like Tor, this delay can introduce latency in one logical stream even though the loss may be in a different stream.

In this work, we attempt to reduce the observed latency in Tor streams by eliminating cross-stream head-of-line blocking. Unordered TCP (uTCP) and Unordered TLS (uTLS) [20] have been shown to combat head-of-line blocking in cases where logical streams are multiplexed over TCP by enabling out-of-order delivery of packets upon the application's request. By deploying uTCP and uTLS at relay servers within the Tor network, we expect to reduce the interdependence created by interleaving streams.

The code changes are minimal, though they do affect multiple layers of the stack. We deploy uTCP and uTLS to support unordered deliveries at the OS and socket-level, respectively, each with a small ($< 5\%$) code increase. The application-level changes to Tor are even smaller (95 lines, $\ll 1\%$). Although we have not performed an extensive experimental evaluation, our initial experiments suggest the feasibility and benefits of our approach. Tor presents a unique deployment scenario for uTCP and uTLS, which primarily focus on application-perceived latency. In Tor, unlike other applications, reducing latency cannot come at the cost of its anonymity or security properties. Thus, the isolation of changes just to delivery semantics, leaving other properties such as congestion control and wire format unaffected, makes uTCP and uTLS an ideal protocol stack for this situation.

2 Motivation

This section explores the design of the Tor network and its use of TCP as a transport substrate. We then illustrate the head-of-line blocking problem inherent to any application using TCP in this way. Lastly, we discuss related work and propose a set of requirements for a solution.

2.1 The Tor Network

Tor [10] is a distributed overlay network and protocol for carrying generic application data, though it is often used for web browsing. Tor enhances a user's privacy by interposing several additional hops between the user's client node and a desired communication target (e.g., web server) external to the Tor network. This indirection prevents the server from identifying the client directly.

The client uses layered symmetric-key encryption on the payload to prevent eavesdropping while data is en route. The client encrypts upstream payloads with several layers of encryption; each intermediate Onion Router (OR) then decrypts one layer before relaying each payload. Downstream traffic from the server follows the reverse path, with each OR adding a layer of encryption to each payload and the client decrypting all layers.

A Directory Server (DS) dynamically maintains a set of ORs that are willing to act as relays for communi-

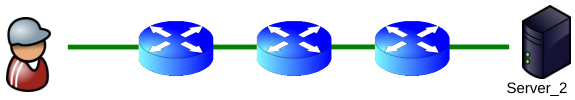

Figure 1: A simple Tor circuit using 3 Onion Routers.

cation. A user then executes the Onion Proxy (OP) software locally, which contacts a DS and selects a set (usually three) of ORs for the current session. The user then passes application data (e.g., HTTP [5, 12] request/response) through the OP, where they follow the pre-built *circuit* through the Tor network and exit before reaching the desired endpoint. Figure 1 shows a diagram of an example Tor circuit.

In addition to encrypting payloads manually, Tor uses TLS [9] to encrypt OR-to-OR connections. From TLS downward, Tor follows the traditional protocol stack for encrypted web traffic: application data over TLS, over TCP [27], over IP [17]. A beneficial side effect of using this traditional stack is that, apart from sophisticated packet size and timing analysis, Tor traffic appears similar to common, "innocuous" e-commerce traffic in transit. A potentially negative consequence of using this TLS/TCP stack, however, is its delivery semantics.

2.2 A Pure TCP Substrate

TCP is a stable and robust protocol, thanks to decades of evolution and fine-tuning, and it is by far the most popular transport used in the Internet. Partly as a result of its popularity and familiarity, many applications use TCP even when other transport semantics may fit better. TCP's strict in-order delivery model is ill-suited for real-time applications and other applications in which there is a natural division of data into multiple logically independent communication units or streams. TCP enforces serialization of data on the wire and only delivers the data in the same order at the receiver. When logically distinct streams are sharing an underlying TCP connection, this serialization is often arbitrary and unimportant to the application. TCP, however, delays delivering a packet until all prior packets have been delivered, causing a lost packet on one logical stream to delay all other streams sharing the connection unnecessarily.

Tor for example uses point-to-point TCP connections between relays, serializing packets from multiple Tor circuits onto a single TCP connection for each pair of communicating relays. Thus, a dropped packet between two relays will delay the forwarding of packets on its own *and all other* circuits sharing this OR-to-OR link. This is known as the head-of-line blocking problem. Figure 2 shows a diagram of this situation in Tor with 3 circuits sharing a single point-to-point connection.

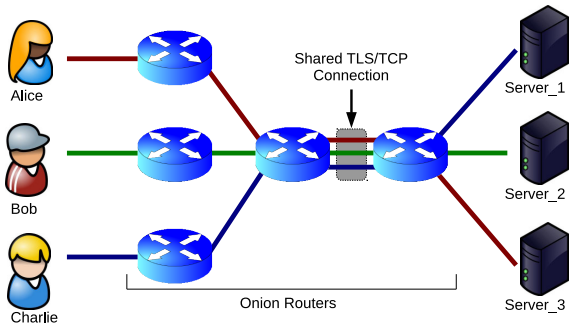

Figure 2: Tor circuits sharing a single TCP connection, creating a head-of-line blocking situation across circuits.

2.3 TCP Substrate Proliferation

Head-of-line blocking is well-known and understood and, in many cases, tolerated simply because alternatives can be more difficult to deploy. Despite the fact that other transports, including UDP [22] and SCTP [25], support unordered delivery, applications consistently tend to build their own transports atop TCP. Examples include media transports such as RTP [24], and experimental multi-streaming transports such as SST [13], SPDY [2], and ØMQ [1]. Applications increasingly use HTTP or HTTPS over TCP as a substrate [21]. Furthermore, a recent study found over 70% of streaming media using TCP [15], and even latency-sensitive conferencing applications such as Skype often use TCP [4].

Despite sometimes non-ideal delivery semantics, TCP also offers other real benefits over UDP and other transports, including: robust congestion control, fine-tuned timeout and retransmissions, flow control, and benefits from Performance-Enhancing-Proxies [6] and other modern hardware optimizations for TCP.

Even if one wishes to use TCP for compatibility or reachability, one could break the interdependence on the wire by using separate TCP connections for each stream (or circuit). However, this approach incurs considerable network latency and bandwidth and host memory costs for each TCP connection, may consume many socket file descriptors, and may exhaust the port space for popular Tor relays, potentially discouraging casual volunteers.

2.4 Goals and Related Work

There have been previous attempts to solve Tor's head-of-line blocking problem. We briefly describe a few in light of Tor's overall goals and requirements. Reardon's approach tunnels Tor's TCP traffic over DTLS [23], effectively breaking interdependence on the wire via UDP's unordered delivery. This approach relies on a minimalist userspace TCP stack and, along with others using UDP [19], results in a different wire format and significant code changes. DefenstraTor [3] addresses the latency issue by changing the shared congestion control

context between circuits, but does not address the head-of-line blocking problem. Our work preserves Tor's approach to congestion control: multiple circuits on the same TCP connection share a congestion control context. Torchestra [14] and other prioritization schemes [26] reduce latency on interactive flows by separating them from bulk downloads. These approaches generally propose significant changes to Tor and its protocol and do not actually eliminate head-of-line blocking for circuits on the same TCP connection.

Our approach to the head-of-line blocking problem attempts to satisfy the following requirements. First, the solution must be fully backward compatible, so as not to alienate the established Tor user base. Ideally, we envision Tor relays negotiating a new OR-to-OR protocol version, and automatically employing the solution within the Tor network with no externally visible changes to clients. Second, the solution should be modular enough that two relays could negotiate its use completely on their own. This allows for an incremental rollout with better testing and enables individual relay operators to monitor and evaluate the effects of an upgrade. Lastly, eliminating head-of-line blocking should not weaken the privacy and anonymity properties of the protocol.

3 TCP in Tor

This section explores how Tor uses TCP as its underlying transport substrate. We examine how Tor constructs circuits, why Tor multiplexes them over a single TCP connection, and why this design is problematic for latency in the presence of packet loss.

3.1 Tor Components

As previous introduced, there are three types of nodes in the Tor network: 1) Directory Server (or Authority) (DS) – maintains lists of available Onion Routers and connectivity status; 2) Onion Router (OR) – acts as a relay server, accepting cells, performing enryption or decryption, and routing them to the next hop in the circuit; 3) Onion Proxy (OP) – client-executed proxy server running locally on the client's machine, responsible for obtaining lists of ORs from a DS and injecting requests into the Tor network along negotiated circuits.

DSs and ORs need to be more stable nodes – volunteers launch instances and may let them run for weeks or longer; while OPs are more fleeting – a client may launch an OP for a single browsing session and then terminate the process afterward. Additionally, OPs only participate as the "origin" of circuits; they do not act as relays for other users' circuits.

On startup, ORs establish TCP connections to multiple DSs to advertise their willingness to relay traffic. Occasionally, an OR will perform a bandwidth assessment test, or transmit other types of "heartbeat" communication back to the DS. Similarly, an OP contacts multiple DSs and obtains information about the currently available ORs and their properties (location, bandwidth, etc.).

For this control traffic, TCP is an appropriate transport as the reliability and in-order delivery provide essential guarantees for correctness. Additionally, TCP's widespread support throughout the Internet helps maximize reachability and connectivity within the Tor network. For example, some NATs [11] might block non-TCP traffic making it difficult to deploy Tor and other application overlays over another transport such as UDP.

3.2 Tor Circuits

A circuit consists of multiple ORs (usually 3) that ferry traffic sent from the OP. The final OR in the circuit establishes a direct TCP connection with the OP's desired destination, often external to Tor. Each request a client sends to its local OP is broken into *cells* that are relayed from OR to OR along the OP's pre-determined circuit. The OP encrypts each cell multiple times prior to transmission, forming an "onion skin" several layers thick around the data. Each OR in the circuit decrypts one encryption layer, until the final OR produces the client's original request and transmits it to the external endpoint.

An OP is responsible for selecting ORs for a circuit from the list of available ORs that it receives from the DSs. An OP is free to discriminate among ORs based on geolocation or other arbitrary attributes such as bandwidth or past performance. (Tor's configuration file format allows for this discrimination.) This requirement is somewhat due to the sensitive nature of Tor's anonymous communication: a user may not wish to route its traffic through a particular OR if he perceives that OR is potentially under control of a malicious or adversarial entity.

To hide the origin's identity, the OP never directly contacts any OR other than the first OR in the circuit. When building a circuit, the OP obtains from the DS the public keys of all ORs selected for the circuit. During circuit initialization, the ORs use an "extend" message to the next OR in a circuit, establishing point-to-point connections to carry the OP's data. This design decision to have ORs act on behalf of the OP to establish each additional link in the circuit is fundamental to Tor's security model.

The potential volume of circuits in flight through a single OR at any moment prevents ORs from using a new TCP connection per circuit. This is partly due to the port exhaustion problem discussed earlier. Additionally, TCP's 3-way handshake per connection has been shown to increase latency due to roundtrip costs [2, 13].

Thus, a single TCP connection carries all cells for all circuits traveling between two ORs. This design decision amplifies TCP's delivery semantics all the way into the application's processing loop. Because of TCP's strict in-order delivery, application datagrams can only be pro-

cessed in the order that they were sent over the network, regardless of when they actually arrive at the destination.

The cross-circuit interdependence introduced by TCP manifests itself in the form of head-of-line blocking, and increases latency for circuits even if the circuit's data itself was not dropped. Whether or not reliability and in-order delivery are the correct design decisions *within* circuits is not discussed here – that is the abstraction that Tor's designers chose. We do not argue for changing this abstraction, as that would require a much bigger overhaul, especially to the application endpoints that would then need to handle unordered application data. Instead, we argue for simply breaking the interdependence *across* circuits that TCP inherently introduces.

4 Unordered TCP and TLS

There are potentially many designs to eliminate head-of-line blocking in Tor circuits. Here, we describe just one approach using two techniques for delivering unordered data with TCP in a wire-compatible way.

4.1 Unordered TCP

Unordered TCP (uTCP) [20] is a technique used for delivering unordered TCP data in a wire-compatible way. We describe the high-level design points here for clarity.

The goal of uTCP is to allow applications to request immediate delivery of available data, even if the data is not the "next" data in the TCP byte sequence space. The uTCP code is a kernel patch that provides a new socket option for enabling unordered delivery. When set, a call to `read()` returns a contiguous region of bytes, either the next logical bytes in the stream, or some future region in the stream (i.e. out-of-order). Additionally, each call to `read()` returns metadata indicating the position of the contiguous bytes in the TCP byte sequence space. With this metadata, the application, or another library [7, 9], can reorder the segments as needed using application-level "framing markers", for example.

The design of uTCP is motivated by the goals of maintaining exact compatibility with TCP's existing wire-visible protocol and behavior, while simply exposing data that the kernel normally buffers. As discussed previously, exact wire compatibility is important for Tor, and the reduction of time spent in kernel buffers should expedite the relaying of cells.

4.2 Unordered TLS

Unordered TLS (uTLS) provides traditional TLS-like [9] security atop *Unordered TCP*. Though uTLS is not new, we describe its high-level design here for clarity.

The goal of uTLS (like uTCP) is to remain entirely indistinguishable on the wire from regular TLS. To this end, uTLS uses the metadata returned by uTCP on each socket read to construct an ordered queue of the byte stream at the receiver. It then scans this byte stream looking for the 5-byte TLS record header. If it finds what it believes to be a record, it attempts to decrypt and authenticate the record. If authentication succeeds, the record is returned. Otherwise, the bytes remain queued, eventually to be decrypted in-order once any "holes" are filled.

Because the uTLS code is simply a receiver-side change, the wire format is unaffected, and appears identical to HTTPS when run on port 443, regardless of whether the encrypted data carries HTTP headers or not. This is especially useful for Tor as HTTP is not the only application-level protocol Tor supports. Furthermore, uTLS preserves the security properties of TLS. uTLS requires TLS protocol version 1.2 in CTR or CBC mode, with explicit initialization vectors in CBC mode.

5 Unordered Tor

Given the techniques described above and that Tor already communicates in a datagram-oriented way, we made few significant code changes to support unordered cell processing. Although we did change Tor's wire format, ORs negotiate this using a new Tor version number and fall back to a previous version if either OR does not support "Unordered Tor". We now summarize the changes we made to the Tor source code.

5.1 Cell Boundaries

Processing cells out-of-order over TCP requires that the underlying substrate provide Tor with complete cells on each socket read. Otherwise, Tor may process portions of cells as though they were complete cells, causing failure. Fortunately, uTLS already creates records on the byte boundaries it receives at the time of the sender's `write()` call. These boundaries are preserved over the wire, and the receiver's subsequent `read()` returns the same byte regions. Thus, if the Tor sender writes complete cells into the uTLS socket, it is guaranteed that the receiver will read byte regions on cell boundaries.

Modifying Tor to produce this behavior was straightforward. We ensure that each time a cell is ready for transmission, it triggers a call to `write()` on the socket. Similarly, Tor already tries to process data after each read, optionally reading more data if it had an incomplete cell. With complete cells being written and uTLS boundary preservation, each read now produces a complete cell, ready for processing.

5.2 Sequence Numbers and Cell Packing

To guarantee correctness, we wish to process cells in-order within a circuit. To identify the ordering of cells within a circuit, we added a sequence number to all outgoing cells. This required a change to Tor's wire format, which we now negotiate with a new Tor version number.

ORs maintain sequence numbers for each circuit in both the "forward" and "backward" directions, from the origin and to the origin, respectively. The transmitting OR writes the sequence number into each cell header upon transmission. Additionally, ORs maintain an "expected" sequence number for each direction on the circuit. The receiving OR then increments the circuit's expected sequence number whenever an in-order packet arrives on that circuit. This exposes a *circuit-level* granularity for ordering cells, rather than a *connection-level* granularity, as is the case currently in Tor. We use a 16-bit sequence number, but 32-bits could be used if needed. We perform no special handling for sequence number wrap-around, but leave this to future work.

The extra 2 bytes for the sequence number in the cell header increase the cell header size from 5 to 7 bytes, leaving the cell payload size of 509 bytes unchanged. (The header is 9 bytes if using "wide" circuit ids.) By changing only the header, an OR can receive a cell from a normal TCP connection and send out the same payload with the new "unordered" header. Because cells are processed in-order *within* a circuit, a TCP-based OR will never receive cells out-of-order within a circuit, even if the transmitting uTCP-based OR processed cells across circuits out-of-order. In this way, two ORs can agree to use the new header format independently (and benefit from unordered delivery), without consent from or negative effect to other nodes on the circuit. Simply put, a transmitting OR uses the header format it negotiated with the receiving OR at connection setup.

With a sequence number added to cells, we needed to adjust Tor's wire-packing code. This code serializes cell headers into network order and produces a byte array for transmission, performing the opposite task on reception. We modified Tor to correctly serialize and deserialize cells containing a sequence number.

5.3 In-Order Circuits

Tor already uses a cell queue for storing received cells prior to processing, although we find that in practice the queue rarely fills up. We leveraged this existing queue to store cells that arrive out-of-order within their circuit.

If a cell's sequence number matches the circuit's "expected" sequence number, we process the cell normally and increment the expected sequence number. Otherwise, the OR places the cell in Tor's receive queue to be processed once the circuit's expected sequence number has increased to match the cell's. We thus enforce in-order delivery within each circuit to avoid breaking circuit-level protocol invariants, for example, by processing a DESTROY cell prior to a data cell on that circuit.

	Tor	Unordered Tor	Delta
Lines of Code	81418	81513	+95 (0.001%)

Table 1: Code size of Unordered Tor prototype as a delta compared to stock Tor for the OR code only.

5.4 Correctness

As just stated, Unordered Tor does not currently support unordered delivery within a circuit: all cells are processed in the same order within a circuit as in normal Tor. Future work might investigate mechanisms to handle unordered data within a circuit, which might benefit the performance of UDP-based applications tunneled over TOR, for example. For now, however, we prioritize simplicity, correctness, and ease of deployment.

Similarly, the per-cell onion encryption is unaffected because cells within the same circuit are processed in-order. A Tor node maintains a separate encryption context for each circuit, ensuring that cell encryption and decryption only affects the circuit the cell belongs to. At the transport level, uTLS [20] demonstrated that there is no loss of security in delivering TLS records out-of-order.

However, we do not perform a formal and thorough security analysis here, and leave to future work the investigation of whether Unordered Tor might expose new vulnerabilities, such as traffic analysis or timing attacks.

6 Preliminary Evaluation

We provide a simple proof-of-concept experiment to validate that Unordered Tor" can address head-of-line blocking and reduce latency across circuits. We believe this experiment is suggestive of the results achievable in a large-scale authentic deployment of Unordered Tor.

6.1 Prototype Implementation

Our prototype implements the changes outlined in Section 5. To measure code complexity, we compare the lines of code [8] in stock Tor and Unordered Tor for the OR code only (not the tools or proxy).

Table 1 shows that the code complexity for enabling unordered relaying in Tor is extremely small – much less than 1%. This small delta is due mainly to the fact that Tor already processes application payloads in datagram-like cells, enabling us to reuse existing data structures.

6.2 Experimental Setup

Our test setup uses a small virtual Tor network to create a controlled environment to examine the behavior of Unordered Tor. We make no claims that this experiment portrays Tor's performance "in the wild" on the real Internet. Instead, we merely use this setup to highlight the differences in behavior between the two variants.

Our Tor testbed network contains 3 relay servers (ORs), 3 Directory Authorities, and a single proxy (OP)

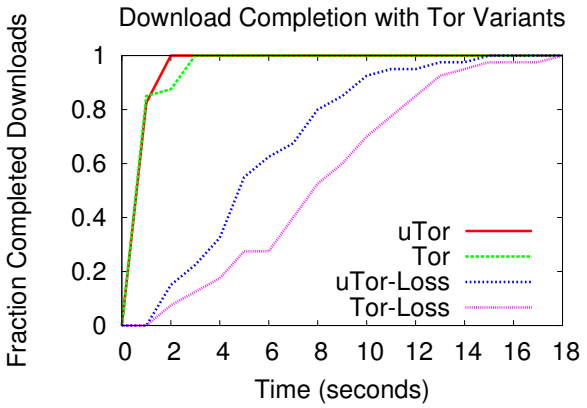

Figure 3: Application-observed completion times for HTTP downloads using Tor relays with and without unordered delivery.

instance. We configure the OP to use the same path (same entry, middle, and exit relays) for all circuits. This setup creates competing traffic, and enables us to induce head-of-line blocking across circuits by triggering loss on a single link. Each client request to the OP is routed through the Tor network, and then exits the network to the desired endpoint on the external Internet.

The Tor network is configured with a normally distributed, 50ms mean one-way propagation delay. For the series names ending in "-Loss", the path from the second OR to the exit (i.e. final) OR was configured with a 5% artificial loss rate on the outgoing link. We acknowledge this high loss rate is unrealistic, but use it to emphasize the difference between stock Tor and Unordered Tor. We believe even the more moderate loss rates common in the Internet may still affect a significant number of users, given the large size of Tor's user base and the heavy traffic loads that popular Tor ORs commonly sustain.

6.3 Results

Figure 3 presents the application-observed completion time for a set of simultaneous, independent HTTP downloads. As mentioned above, both Tor and Unordered Tor (uTor) were tested twice: with and without artificial loss on the link to the exit relay. The experiment issues web requests to 40 popular sites on the Internet, with the download sizes ranging from 10KB to 400KB.

Under normal network conditions, we see that Tor and uTor perform similarly in over 80% of downloads, which is expected for the many (often short) downloads that encounter no TCP segment loss event. In about 15% of downloads, however, uTor shows significantly lower latency – often 30–40% – by eliminating head-of-line blocking and localizing to a single Tor circuit the latency impact of normal, occasional TCP segment losses.

The artificial loss cases amplify this difference, showing a much larger effect on flow completion times. In uTor, over 50% of flows complete within 5 seconds, while less than 30% of flows over stock Tor complete in the same time span. This illustrates the head-of-line blocking scenario: with drops, Tor must delay all flows while uTor allows unaffected circuits to proceed. Both uTor and Tor show some "stragglers" – flows that do not complete until long after all others. This is expected, as unordered delivery does not eliminate drops, only their influence on subsequent data on the same TCP link.

6.4 Future Work

Our next task is to deploy and evaluate Unordered Tor in more real-world scenarios, to test interoperability on circuits with both uTCP and TCP hops, and to investigate more subtle protocol issues such as negotiation fall-back and sequence number wrap-around. We intend to open-source our code and share it with the Tor development community once the prototype is sufficiently mature.

In the longer term, we would like to explore Tor and application extensions to process cells in a single circuit out-of-order, which may benefit UDP-based VoIP or VPN applications for example. We also wish to explore further the prevalence of head-of-line blocking in other applications that do stream multiplexing, such as SPDY [2]. While it is logically clear how latency follows from in-order semantics, we have limited quantitative data on how high this cost is in practice.

7 Conclusion

In this work, we demonstrate and analyze the existence of head-of-line blocking in Tor. We show how the integration of two techniques known to reduce head-of-line blocking in TCP can alleviate application-perceived latency in Tor circuits. We show that the changes to the Tor source are minimal and fully compatible both with previous versions of Tor and the goals of Tor to subvert exposure and disruption. Although a full-scale evaluation is needed to prove the feasibility of this deployment scenario, we believe any application using TCP as a multiplexing substrate can benefit from our approach.

Acknowledgments

We thank Nick Mathewson for his help integrating our code in the Tor network, and the anonymous reviewers for their valuable feedback. This research was conducted with Government support under and awarded by DoD, Air Force Office of Scientific Research, and National Defense Science and Engineering Graduate (NDSEG) Fellowship, 32 CFR 168a. Additionally, this material is based upon work supported by the Defense Advanced Research Agency (DARPA) and SPAWAR Systems Center Pacific, Contract No. N66001- 11-C-4018.

References

[1] ØMQ: The intelligent transport layer. http://www.zeromq.org.

[2] SPDY: An experimental protocol for a faster Web. http://www.chromium.org/spdy/spdy-whitepaper.

[3] Mashael AlSabah, Kevin Bauer, Ian Goldberg, Dirk Grunwald, Damon McCoy, Stefan Savage, and Geoffrey Voelker. DefenestraTor: Throwing out windows in Tor. In *11th Privacy Enhancing Technologies (PETS)*. July 2011.

[4] Salman A. Baset and Henning Schulzrinne. An analysis of the Skype peer-to-peer Internet telephony protocol. In *IEEE INFOCOM*, April 2006.

[5] T. Berners-Lee, R. Fielding, and H. Frystyk. Hypertext transfer protocol – HTTP/1.0, May 1996. RFC 1945.

[6] Brian Carpenter and Scott Brim. Middleboxes: Taxonomy and Issues, February 2002. RFC 3234.

[7] Stuart Cheshire and Mary Baker. Consistent Overhead Byte Stuffing. In *ACM SIGCOMM*, September 1997.

[8] Al Danial. Counting Lines of Code, ver. 1.53. http://cloc.sourceforge.net/.

[9] T. Dierks and E. Rescorla. The transport layer security (TLS) protocol version 1.2, August 2008. RFC 5246.

[10] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: the second-generation onion router. In *USENIX Security Symposium*, August 2004.

[11] K. Egevang and P. Francis. The IP network address translator (NAT), May 1994. RFC 1631.

[12] R. Fielding et al. Hypertext transfer protocol – HTTP/1.1, June 1999. RFC 2616.

[13] Bryan Ford. Structured streams: a new transport abstraction. In *ACM SIGCOMM*, August 2007.

[14] Deepika Gopal and Nadia Heninger. Torchestra: reducing interactive traffic delays over Tor. In *Workshop on Privacy in the Electronic Society (WPES)*, October 2012.

[15] Lei Guo, Enhua Tan, Songqing Chen, Zhen Xiao, Oliver Spatscheck, and Xiaodong Zhang. Delving into Internet streaming media delivery: a quality and resource utilization perspective. In *Internet Measurement Conference (IMC)*, October 2006.

[16] Amir Houmansadr, Chad Brubaker, and Vitaly Shmatikov. The parrot is dead: Observing unobservable network communications. In *IEEE Security and Privacy*, May 2013.

[17] Internet protocol, September 1981. RFC 791.

[18] Hooman Mohajeri Moghaddam, Baiyu Li, Mohammad Derakhshani, and Ian Goldberg. SkypeMorph: Protocol obfuscation for Tor bridges. In *ACM Conference on Computer and Communications Security (CCS)*, October 2012.

[19] Steven J. Murdoch. Comparison of Tor datagram designs. Technical Report 2011-11-001, The Tor Project, November 2011.

[20] Michael F. Nowlan, Nabin Tiwari, Janardhan Iyengar, Syed Obaid Amin, and Bryan Ford. Fitting square pegs through round pipes: Unordered delivery wire-compatible with TCP and TLS. April 2012.

[21] Lucian Popa, Ali Ghodsi, and Ion Stoica. HTTP as the narrow waist of the future Internet. In *9th ACM Workshop on Hot Topics in Networks (HotNets-IX)*, October 2010.

[22] J. Postel. User datagram protocol, August 1980. RFC 768.

[23] Joel Reardon and Ian Goldberg. Improving tor using a TCP-over-DTLS tunnel. In *18th USENIX Security Symposium*, August 2009.

[24] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. RTP: A transport protocol for real-time applications, July 2003. RFC 3550.

[25] R. Stewart, ed. Stream control transmission protocol, September 2007. RFC 4960.

[26] Can Tang and Ian Goldberg. An improved algorithm for Tor circuit scheduling. In *17th ACM conference on Computer and Communications Security (CCS)*, October 2010.

[27] Transmission control protocol, September 1981. RFC 793.

[28] Zachary Weinberg, Jeffrey Wang, Vinod Yegneswaran, Linda Briesemeister, Steven Cheung, Frank Wang, and Dan Boneh. StegoTorus: a camouflage proxy for the Tor anonymity system. In *19th ACM conference on Computer and Communications Security (CCS)*, October 2012.