Improving Tor using a TCP-over-DTLS Tunnel

Joel Reardon*
Google Switzerland GmbH
Brandschenkestrasse 110
Zürich, Switzerland
reardon@google.com

Ian Goldberg
University of Waterloo
200 University Ave W.
Waterloo, ON, Canada
iang@cs.uwaterloo.ca

Abstract

The Tor network gives anonymity to Internet users by relaying their traffic through the world over a variety of routers. All traffic between any pair of routers, even if they represent circuits for different clients, are multiplexed over a single TCP connection. This results in interference across circuits during congestion control, packet dropping and packet reordering. This interference greatly contributes to Tor's notorious latency problems.

Our solution is to use a TCP-over-DTLS (Datagram Transport Layer Security) transport between routers. We give each stream of data its own TCP connection, and protect the TCP headers—which would otherwise give stream identification information to an attacker—with DTLS. We perform experiments on our implemented version to illustrate that our proposal has indeed resolved the cross-circuit interference.

1 Introduction

Tor [2] is a tool to enable Internet privacy that has seen widespread use and popularity throughout the world. Tor consists of a network of thousands of nodes—known as Onion Routers (ORs)—whose operators have volunteered to relay Internet traffic around the world. Clients—known as Onion Proxies (OPs)—build circuits through ORs in the network to dispatch their traffic. Tor's goal is to frustrate an attacker who aims to match up the identities of the clients with the actions they are performing. Despite its popularity, Tor has a problem that dissuades its ubiquitous application—it imposes greater latency on its users than they would experience without Tor.

While some increased latency is inevitable due to the increased network path length, our experiments show that this effect is not sufficient to explain the increased cost. In Section 2 we look deeper, and find a component of the transport layer that can be changed to improve Tor's performance. Specifically, each pair of routers maintains a single TCP connection for all traffic that is sent between them. This includes multiplexed traffic for different circuits, and results in cross-circuit interference that degrades performance. We find that congestion control mechanisms are being unfairly applied to all circuits when they are intended to throttle only the noisy senders. We also show how packet dropping on one circuit causes interference on other circuits.

Section 3 presents our solution to this problem—a new transport layer that is backwards compatible with the existing Tor network. Routers in Tor can gradually and independently upgrade, and our system provides immediate benefit to any pair of routers that choose to use our improvements. It uses a separate TCP connection for each circuit, but secures the TCP header to avoid the disclosure of per-circuit data transfer statistics. Moreover, it uses a user-level TCP implementation to address the issue of socket proliferation that prevents some operating systems from being able to volunteer as ORs.

Section 4 presents experiments to compare the existing Tor with our new implementation. We compare latency and throughput, and perform timing analysis of our changes to ensure that they do not incur non-negligible computational latency. Our results are favourable: the computational overhead remains negligible and our solution is successful in addressing the improper use of congestion control.

Section 5 compares our enhanced Tor to other anonymity systems, and Section 6 concludes with a description of future work.

1.1 Apparatus

Our experiments were performed on a commodity Thinkpad R60—1.66 GHz dual core with 1 GB of RAM. Care was taken during experimentation to ensure that the system was never under load significant enough to influ-

*Work done while at the University of Waterloo

ence the results. Our experiments used a modified version of the Tor 0.2.0.x stable branch code.

2 Problems with Tor's Transport Layer

We begin by briefly describing the important aspects of Tor's current transport layer. For more details, see [2]. An end user of Tor runs an Onion Proxy on her machine, which presents a SOCKS proxy interface [7] to local applications, such as web browsers. When an application makes a TCP connection to the OP, the OP splits it into fixed-size *cells* which are encrypted and forwarded over a *circuit* composed of (usually 3) Onion Routers. The last OR creates a TCP connection to the intended destination host, and passes the data between the host and the circuit.

The circuit is constructed with hop-by-hop TCP connections, each protected with TLS [1], which provides confidentiality and data integrity. The OP picks a first OR (OR_1), makes a TCP connection to it, and starts TLS on that connection. It then instructs OR_1 to connect to a particular second OR (OR_2) of the OP's choosing. If OR_1 and OR_2 are not already in contact, a TCP connection is established between them, again with TLS. If OR_1 and OR_2 *are* already in contact (because other users, for example, have chosen those ORs for their circuits), the existing TCP connection is used for all traffic between those ORs. The OP then instructs OR_2 to contact a third OR, OR_3, and so on. Note that there is *not* an end-to-end TCP connection from the OP to the destination host, nor to any OR except OR_1.

This multi-hop transport obviously adds additional unavoidable latency. However, the observed latency of Tor is larger than accounted for simply by the additional transport time. In [12], the first author of this paper closely examined the sources of latency in a live Tor node. He found that processing time and input buffer queueing times were negligible, but that output buffer queueing times were significant. For example, on an instrumented Tor node running on the live Tor network, 40% of output buffers had data waiting in them from 100 ms to over 1 s more than 20% of the time. The data was waiting in these buffers because the the operating system's output buffer for the corresponding socket was itself full, and so the OS was reporting the socket as unwritable. This was due to TCP's congestion control mechanism, which we discuss next.

Socket output buffers contain two kinds of data: packet data that has been sent over the network but is unacknowledged[1], and packet data that has not been sent due to TCP's congestion control. Figure 1 shows the size of the socket output buffer over time for a particular connection. First, unwritable sockets occur when the remaining capacity in an output buffer is too small to accept new data. This in turn occurs because there is already too much data in the buffer, which is because there is too much unacknowledged data in flight and throttled data waiting to be sent. The congestion window (CWND) is a variable that stores the number of packets that TCP is currently willing to send to the peer. When the number of packets in flight exceeds the congestion window then the sending of more data is throttled until acknowledgments are received. Once congestion throttles sending, the data queues up until either packets are acknowledged or the buffer is full.

In addition to congestion control, TCP also has a flow control mechanism. Receivers advertise the amount of data they are willing to accept; if more data arrives at the receiver before the receiving application has a chance to read from the OS's receive buffers, this advertised receiver window will shrink, and the sender will stop transmitting when it reaches zero. In none of our experiments did we ever observe Tor throttling its transmissions due to this mechanism; the advertised receiver window sizes never dropped to zero, or indeed below 50 KB. Congestion control, rather than flow control, was the reason for the throttling.

While data is delayed because of congestion control, it is foolhardy to attempt to circumvent congestion control as a means of improving Tor's latency. However, we observe that Tor's transport between ORs results in an *unfair* application of congestion control. In particular, Tor's circuits are multiplexed over TCP connections; i.e., a single TCP connection between two ORs is used for multiple circuits. When a circuit is built through a pair of unconnected routers, a new TCP connection is established. When a circuit is built through an already-connected pair of ORs, the existing TCP stream will carry both the existing circuits and the new circuit. This is true for all circuits built in either direction between the ORs.

In this section we explore how congestion control affects multiplexed circuits and how packet dropping and reordering can cause interference across circuits. We show that TCP does not behave optimally when circuits are multiplexed in this manner.

2.1 Unfair Congestion Control

We believe that multiplexing TCP streams over a single TCP connection is unwise and results in the unfair application of TCP's congestion control mechanism. It results in multiple data streams competing to send data over a TCP stream that gives more bandwidth to circuits that send more data; i.e., it gives each byte of data the same priority regardless of its source. A busy circuit that

[1]Recall that TCP achieves reliability by buffering all data locally until it has been acknowledged, and uses this to generate retransmission messages when necessary

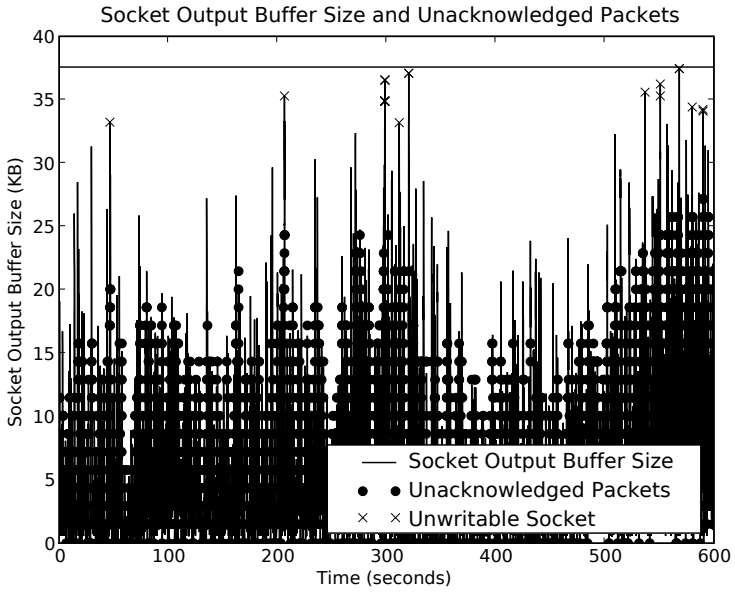

Figure 1: TCP socket output buffer size, writability, and unacknowledged packets over time.

triggers congestion control will cause low-bandwidth circuits to struggle to have their data sent. Figure 2 illustrates data transmission for distinct circuits entering and exiting a single output buffer in Tor. Time increases along the X-axis, and data increases along the Y-axis. The main part of the figure shows two increasing line shapes, each corresponding to the data along a different circuit over time. When the shapes swell, that indicates that Tor's internal output buffer has swelled: the left edge grows when data enters the buffer, and the right edge grows when data leaves the buffer. This results in the appearance of a line when the buffer is well-functioning, and a triangular or parallelogram shape when data arrives too rapidly or the connection is troubled. Additionally, we strike a vertical line across the graph whenever a packet is dropped.

What we learn from this graph is that the buffer serves two circuits. One circuit serves one MB over ten minutes, and sends cells evenly. The other circuit is inactive for the most part, but three times over the execution it suddenly serves 200 KB of cells. We can see that each time the buffer swells with data it causes a significant delay. Importantly, the other circuit is affected despite the fact that it did not change its behaviour. Congestion control mechanisms that throttle the TCP connection will give preference to the burst of writes because it simply provides more data, while the latency for a low-bandwidth application such as `ssh` increases unfairly.

2.2 Cross-Circuit Interference

Tor multiplexes the data for a number of circuits over a single TCP stream, and this ensures that the received data will appear in the precise order in which the component streams were multiplexed—a guarantee that goes beyond what is strictly necessary. When packets are dropped or reordered, the TCP stack will buffer available data on input buffers until the missing in-order component is available. We hypothesize that when active circuits are multiplexed over a single TCP connection, Tor suffers an unreasonable performance reduction when either packet dropping or packet reordering occur. Cells may be available in-order for one particular circuit but are being delayed due to missing cells for another circuit. In-order guarantees are only necessary for data sent within a single circuit, but the network layer ensures that data is only readable in the order it was dispatched. Packet loss or reordering will cause the socket to indicate that no data is available to read even if other circuits have their sequential cells available in buffers.

Figure 3 illustrates the classic head-of-line blocking behaviour of Tor during a packet drop; cells for distinct circuits are represented by shades and a missing packet is represented with a cross. We see that the white, light grey, and black circuits have had all of their data successfully received, yet the kernel will not pass that data to the Tor application until the dropped dark grey packet is retransmitted and successfully received.

We verify our cross-circuit interference hypothesis in two parts. In this section we show that packet drops on a

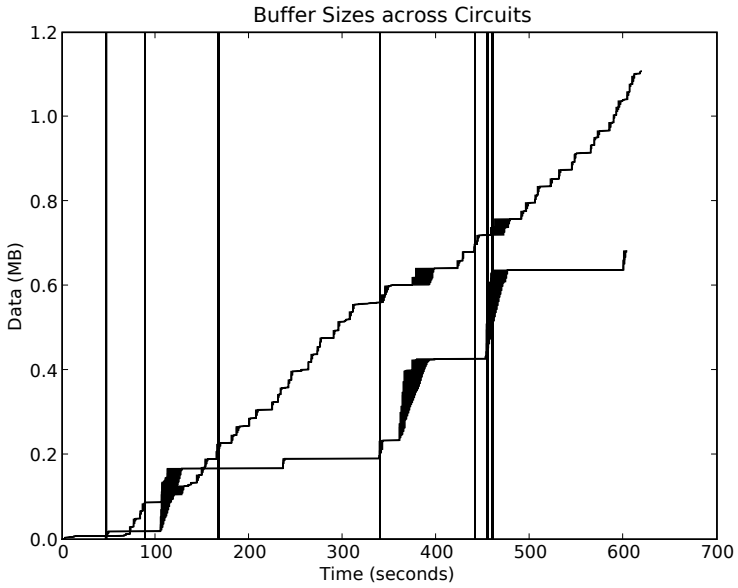

Figure 2: Example of congestion on multiple streams.

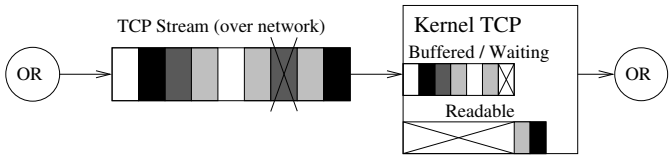

Figure 3: TCP correlated streams. Shades correspond to cells for different circuits.

Experiment 1 Determining the effect of packet dropping on circuit multiplexing.

1: A Tor network of six ORs on a single host was configured to have a latency of 50 milliseconds and a variable packet drop rate.
2: Eight OP built circuits that were fixed so that the second and third ORs were the same for each client, but the first hop was evenly distributed among the remaining ORs. Figure 4 illustrates this setup.
3: There were three runs of the experiment. The first did not drop any packets. The second dropped 0.1% of packets on the shared link, and the third dropped 0.1% of packets on the remaining links.
4: The ORs were initialized and then the clients were run until circuits were established.
5: Each OP had a client connect, which would tunnel a connection to a timestamp server through Tor. The server sends a continuous stream of timestamps. The volume of timestamps measures throughput, and the difference in time measures latency.
6: Data was collected for one minute.

shared link degrade throughput much more severely than drops over unshared links. Then in Section 4 we show that this effect disappears with our proposed solution.

To begin, we performed Experiment 1 to investigate the effect of packet dropping on circuit multiplexing. The layout of circuits in the experiment, as shown in Figure 4, is chosen so that there is one shared link that carries data for all circuits, while the remaining links do not.

In the two runs of our experiments that drop packets, they are dropped according to a target drop rate, either on the heavily shared connection or the remaining connections. Our packet dropping tool takes a packet, decides if it is eligible to be dropped in this experiment, and if so then it drops it with the appropriate probability. However, this model means the two runs that drop packets will see different rates of packet dropping systemwide, since we observe greater traffic on the remaining connections. This is foremost because it spans two hops along the circuit instead of one, and also because traffic from multiple circuits can be amalgamated into one packet for transmission along the shared connection. As a result, a fixed drop rate affecting the remaining connec-

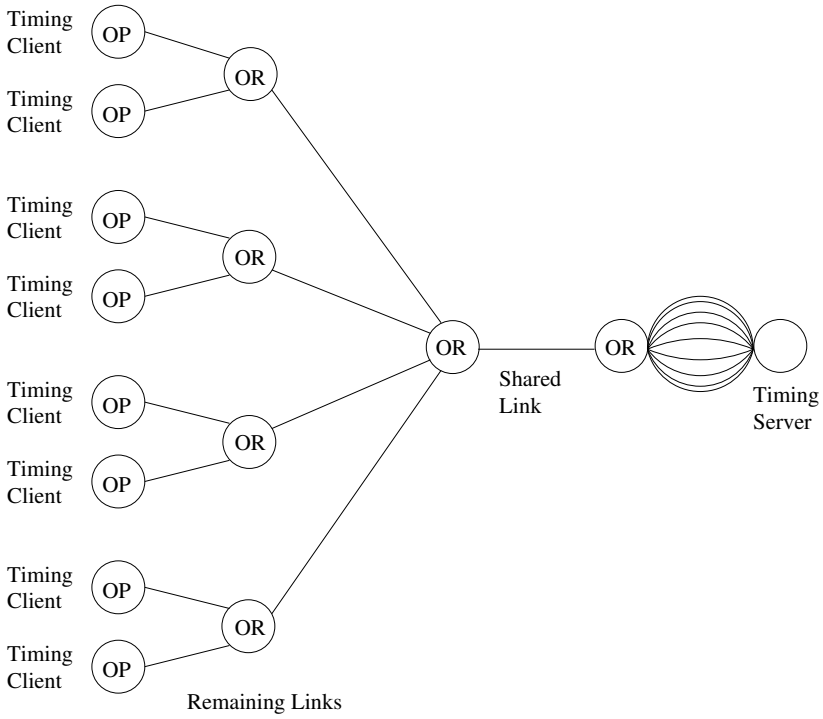

Figure 4: Setup for Experiment 1. The shared link multiplexes all circuits from the various OPs to the final OR; the remaining links carry just one or two circuits each. The splay of links between the final OR and the timing server reflect the fact that a separate TCP connection is made from the final OR to the timing server for each timing client.

Configuration	Network Throughput (KB/s)	Circuit Throughput (KB/s)	Throughput Degradation	Effective Drop Rate
No dropping	221 ± 6.6	36.9 ± 1.1	0 %	0 %
0.1 % (remaining)	208 ± 14	34.7 ± 2.3	6 %	0.08 %
0.1 % (shared)	184 ± 17	30.8 ± 2.8	17 %	0.03 %

Table 1: Throughput for different dropping configurations. Network throughput is the total data sent along all the circuits.

Configuration	Average Latency	Latency Increase	Effective Drop Rate
No dropping	933 ± 260 ms	0 %	0 %
0.1 % (remaining)	983 ± 666 ms	5.4 %	0.08 %
0.1 % (shared)	1053 ± 409 ms	12.9 %	0.03 %

Table 2: Latency for different dropping configurations.

tions will result in more frequent packet drops than one dropping only along the shared connection. This disparity is presented explicitly in our results as the effective drop rate; i.e., the ratio of packets dropped to the total number of packets we observed (including those ineligible to be dropped) in the experiment.

The results of Experiment 1 are shown in Tables 1 and 2. They show the results for three configurations: when no packet dropping is done, when 0.1% of packets are dropped on all connections except the heavily shared one, and when 0.1% of packets are dropped only on the shared connection. The degradation column refers to the loss in performance as a result of introducing packet drops. The average results for throughput and delay were accumulated over half a dozen executions of the experiment, and the mean intervals for the variates are computed using Student's T distribution to 95% confidence.

These results confirm our hypothesis. The throughput degrades nearly threefold when packets are dropped on the shared link instead of the remaining links. This is despite a significantly lower overall drop rate. The behaviour of one TCP connection can adversely affect all correlated circuits, even if those circuits are used to transport less data.

Table 2 suggests that latency increases when packet dropping occurs. Latency is measured by the time required for a single cell to travel alongside a congested circuit, and we average a few dozen such probes. Again we see that dropping on the shared link more adversely affects the observed delay despite a reduced drop rate. However, we note that the delay sees wide variance, and the 95% confidence intervals are quite large.

2.3 Summary

Multiplexing circuits over a single connection is a potential source of unnecessary latency since it causes TCP's congestion control mechanism to operate unfairly towards connections with smaller demands on throughput. High-bandwidth streams that trigger congestion control result in low-bandwidth streams having their congestion window unfairly reduced. Packet dropping and reordering also cause available data for multiplexed circuits to wait needlessly in socket buffers. These effects degrade both latency and throughput, which we have shown in experiments.

To estimate the magnitude of this effect in the real Tor network, we note that 10% of Tor routers supply 87% of the total network bandwidth [8]. A straightforward calculation shows that links between top routers—while only comprising 1% of the possible network links—transport over 75% of the data. At the time of writing, the number of OPs is estimated in the hundreds of thousands and there are only about one thousand active ORs [14]. Therefore, even while most users are idle, the most popular 1% of links will be frequently multiplexing circuits.

Ideally, we would open a separate TCP connection for every circuit, as this would be a more appropriate use of TCP between ORs; packet drops on one circuit, for example, would not hold up packets in other circuits. However, there is a problem with this naive approach. An adversary observing the network could easily distinguish packets for each TCP connection just by looking at the port numbers, which are exposed in the TCP headers. This would allow him to determine which packets were part of which circuits, affording him greater opportunity for traffic analysis. Our solution is to tunnel packets from multiple TCP streams over DTLS, a UDP protocol that provides for the confidentiality of the traffic it transports. By tunnelling TCP over a secure protocol, we can protect both the TCP payload and the TCP headers.

3 Proposed Transport Layer

This section proposes a TCP-over-DTLS tunnelling transport layer for Tor. This tunnel transports TCP packets between peers using DTLS—a secure datagram (UDP-based) transport [9]. A user-level TCP stack running inside Tor generates and parses TCP packets that are sent over DTLS between ORs. Our solution will use a single unconnected UDP socket to communicate with all other ORs at the network level. Internally, it uses a separate user-level TCP connection for each circuit. This decorrelates circuits from TCP streams, which we have shown to be a source of unnecessary latency. The use of DTLS also provides the necessary security and confidentiality of the transported cells, including the TCP header. This prevents an observer from learning per-circuit metadata such data how much data is being sent in each direction for individual circuits. Additionally, it reduces the number of sockets needed in kernel space, which is known to be a problem that prevents some Windows computers from volunteering as ORs. Figure 5 shows the design of our proposed transport layer, including how only a single circuit is affected by a dropped packet.

The interference that multiplexed circuits can have on each other during congestion, dropping, and reordering is a consequence of using a single TCP connection to transport data between each pair of ORs. This proposal uses a separate TCP connection for each circuit, ensuring that congestion or drops in one circuit will not affect other circuits.

3.1 A TCP-over-DTLS Tunnel

DTLS [9] is the datagram equivalent to the ubiquitous TLS protocol [1] that secures much traffic on the Inter-

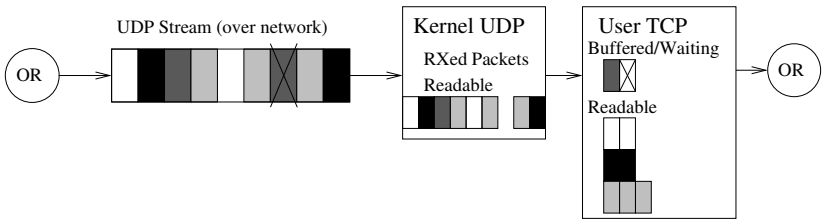

Figure 5: Proposed TCP-over-DTLS Transport showing decorrelated streams. Shades correspond to cells for different circuits (cf. Figure 3).

net today, including https web traffic, and indeed Tor. DTLS provides confidentiality and authenticity for Internet datagrams, and provides other security properties such as replay prevention. IPsec [6] would have been another possible choice of protocol to use here; however, we chose DTLS for our application due to its acceptance as a standard, its ease of use without kernel or superuser privileges, and its existing implementation in the OpenSSL library (a library already in use by Tor). The TLS and DTLS APIs in OpenSSL are also unified; after setup, the same OpenSSL calls are used to send and receive data over either TLS or DTLS. This made supporting backwards compatibility easier: the Tor code will send packets either over TCP (with TLS) or UDP (with DTLS), as appropriate, with minimal changes.

Our new transport layer employs a user-level TCP stack to generate TCP packets, which are encapsulated inside a DTLS packet that is then sent by the system in a UDP/IP datagram. The receiving system will remove the UDP/IP header when receiving data from the socket, decrypt the DTLS payload to obtain a TCP packet, and translate it into a TCP/IP packet, which is then forwarded to the user-level TCP stack that processes the packet. A subsequent read from the user-level TCP stack will provide the packet data to our system.

In our system, the TCP sockets reside in user space, and the UDP sockets reside in kernel space. The use of TCP-over-DTLS affords us the great utility of TCP: guaranteed in-order delivery and congestion control. The user-level TCP stack provides the functionality of TCP, and the kernel-level UDP stack is used simply to transmit packets. The secured DTLS transport allows us to protect the TCP header from snooping and forgery and effect a reduced number of kernel-level sockets.

ORs require opening many sockets, and so our user-level TCP stack must be able to handle many concurrent sockets, instead of relying on the operating system's TCP implementation that varies from system to system. In particular, some discount versions of Windows artificially limit the number of sockets the user can open, and so we use Linux's free, open-source, and high-performance TCP implementation inside user space. Even Windows users will be able to benefit from an improved TCP implementation, and thus any user of an operating system supported by Tor will be able to volunteer their computer as an OR if they so choose.

UDP allows sockets to operate in an unconnected state. Each time a datagram is to be sent over the Internet, the destination for the packet is also provided. Only one socket is needed to send data to every OR in the Tor network. Similarly, when data is read from the socket, the sender's address is also provided alongside the data. This allows a single socket to be used for reading from all ORs; all connections and circuits will be multiplexed over the same socket. When reading, the sender's address can be used to demultiplex the packet to determine the appropriate connection for which it is bound. What follows is that a single UDP socket can be used to communicate with as many ORs as necessary; the number of kernel-level sockets is constant for arbitrarily many ORs with which a connection may be established. This will become especially important for scalability as the number of nodes in the Tor network grows over time. From a configuration perspective, the only new requirement is that the OR operator must ensure that a UDP port is externally accessible; since they must already ensure this for a TCP port we feel that this is a reasonable configuration demand.

Figure 6(a) shows the packet format for TCP Tor, and Figure 6(b) shows the packet format for our TCP-over-DTLS Tor, which has expanded the encrypted payload to include the TCP/IP headers generated by the user-level TCP stack. The remainder of this section will discuss how we designed, implemented, and integrated these changes into Tor.

3.2 Backwards Compatibility

Our goal is to improve Tor to allow TCP communication using UDP in the transport layer. While the original ORs transported cells between themselves, our proposal is to transport, using UDP, both TCP headers and cells between ORs. The ORs will provide the TCP/IP packets to a TCP stack that will generate both the appropriate stream of cells to the Tor application, as well

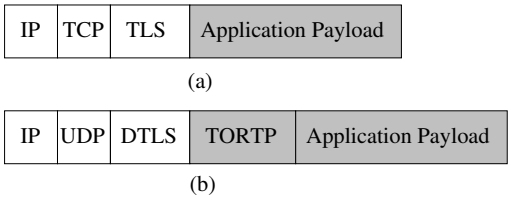

Figure 6: Packets for TCP Tor and our TCP-over-DTLS improved Tor. Encrypted and authenticated components of the packet are shaded in grey. (a) shows the packet format for TCP Tor. (b) shows the packet format for our TCP-over-DTLS Tor. TORTP is a compressed form of the IP and TCP headers, and is discussed in Section 3.4.

as TCP/IP packets containing TCP acknowledgements to be returned.

The integration of this transport layer into Tor has two main objectives. The first is that of interoperability; it is essential that the improved Tor is backwards compatible with the TCP version of Tor so as to be easily accepted into the existing codebase. Recall that Tor has thousands of ORs, a client population estimated in the hundreds of thousands, and has not experienced any downtime since it launched in 2003. It is cumbersome to arrange a synchronized update of an unknown number of anonymous Tor users. A subset of nodes that upgrade and can take advantage of TCP-over-DTLS can provide evidence of the transport's improvement for the user experience—this incremental upgrade is our preferred path to acceptance. Our second objective is to minimize the changes required to the Tor codebase. We must add UDP connections into the existing datapath by reusing as much existing code as possible. This permits future developers to continue to improve Tor's datapath without having to consider two classes of communication. Moreover, it encourages the changes to quickly be incorporated into the main branch of the source code. While it will come with a performance cost for doing unnecessary operations, we perform timing analyses below to ensure that the resulting datapath latency remains negligible.

Interoperability between existing ORs and those using our improved transport is achieved by fully maintaining the original TCP transport in Tor—improved ORs continue to advertise a TCP OR port and multiplexed TCP connections can continue to be made. In addition, improved nodes will also advertise a UDP port for making TCP-over-DTLS connections. Older nodes will ignore this superfluous value, while newer nodes will always choose to make a TCP-over-DTLS connection whenever such a port is advertised. Thus, two UDP nodes will automatically communicate using UDP without disrupting the existing nodes; their use of TCP-over-DTLS is inconsequential to the other nodes. As more nodes support TCP-over-DTLS, more users will obtain its benefits, but we do not require a synchronized update to support our improvements.

Clients of Tor are not required to upgrade their software to obtain the benefits of UDP transport. If two nodes on their circuit use TCP-over-DTLS to communicate then this will happen transparently to the user. In fact, it is important that the user continue to choose their circuit randomly among the ORs: intentionally choosing circuits consisting of UDP nodes when there are only a few such nodes decreases the privacy afforded to the client by rendering their circuit choices predictable.

3.3 User-level TCP Stack

If we simply replaced the TCP transport layer in Tor with a UDP transport layer, our inter-OR communication would then lack the critical features of TCP: guaranteed in-order transmission of streams, and the most well-studied congestion control mechanism ever devised. We wish to remove some of the unnecessary guarantees of TCP for the sake of latency; i.e., we do not need cells from separate circuits over the same connection to arrive in the order they were dispatched. However, we must still be able to reconstruct the streams of each individual circuit at both ends of the connection. We use a TCP implementation in user space (instead of inside the operating system) to accommodate us; a user-level TCP stack provides the implementation of the TCP protocols [4] as part of our program. User-level socket file descriptors and their associated data structures and buffers are accessible only in user space and so are visible and relevant only to Tor. We use the UDP transport layer and DTLS to transport TCP packets between the UDP peers. Only part of the TCP packet is transmitted; the details will be discussed in section 3.4, but it serves our purposes now to conceptualize the two nodes as transporting full TCP packets as the UDP datagram's payload. Upon receiving a UDP datagram, the kernel will remove the UDP header and provide Tor with the enclosed DTLS packet; Tor will decrypt the DTLS payload and present the result (a TCP packet) to its user-level TCP stack. Similarly, when the user-level TCP stack presents a packet for transmission, the node will encrypt it with DTLS and forward the resulting packet to the kernel which then sends it to the

intended destination over UDP. The stack also performs retransmission and acknowledgement of TCP data that are integral to TCP's reliability; these are protected with DTLS and forwarded over UDP in the same manner.

A user-level TCP stack provides an implementation of the suite of socket function calls, such as *socket()*, *send()*, and *recv()*. These reimplementations exist in harmony with the proper set of operating system commands, allowing both a user-level and kernel-level network layer. Thus, data structures and file descriptors created by calls to the user-level stack are visible and relevant only to the parent process; the operating system manages its sockets separately. The user-level stack responds to socket calls by generating packets internally for dispatching as dictated by TCP.

It may seem cumbersome to include an entire TCP implementation as a core component of Tor. In particular, patching the kernel's implementation of TCP to support our features would take significantly less effort. However, Tor relies on volunteers to route traffic; complicated installation procedures are an immediate roadblock towards the ubiquitous use of Tor. The diverse operating systems Tor aims to support and the diverse skill level of its users prevent its installation from requiring external procedures, or even superuser privileges.

Daytona [11] is a user-level TCP stack that we chose for our purposes. It was created by researchers studying network analysis, and consists of the implementation of Linux's TCP stack and the reimplementations of user-level socket functions. It uses `libpcap` to capture packets straight from the Ethernet device and a raw socket to write generated packets, including headers, onto the network. Daytona was designed to operate over actual networks while still giving user-level access to the network implementation. In particular, it allowed the researchers to tune the implementation while performing intrusive measurements. A caveat—there are licensing issues for Daytona's use in Tor. As a result, the deployment of this transport layer into the real Tor network may use a different user-level TCP stack. Our design uses Daytona as a replaceable component and its selection as a user-level TCP stack was out of availability for our proof-of-concept.

3.4 UTCP: Our Tor-Daytona Interface

Our requirements for a user-level TCP stack are to create properly formatted packets, including TCP retransmissions, and to sort incoming TCP/IP packets into data streams: a black box that converts between streams and packets. For our purpose, all notions of routing, Ethernet devices, and interactions with a live network are unnecessary. To access the receiving and transmitting of packets, we commandeer the `rx()` (receive) and `tx()` (transmit) methods of Daytona to instead interface directly with reading and writing to connections in Tor.

UTCP is an abstraction layer for the Daytona TCP stack used as an interface for the stack by Tor. Each UDP connection between ORs has a UTCP-connection object that maintains information needed by our stack, such as the set of circuits between those peers and the socket that listens for new connections. Each circuit has a UTCP-circuit object for similar purposes, such as the local and remote port numbers that we have assigned for this connection.

As mentioned earlier, only part of the TCP header is transmitted using Tor—we call this header the TORTP header; we do this simply to optimize network traffic. The source and destination addresses and ports are replaced with a numerical identifier that uniquely identifies the circuit for the connection. Since a UDP/IP header is transmitted over the actual network, Tor is capable of performing a connection lookup based on the address of the packet sender. With the appropriate connection, and a circuit identifier, the interface to Daytona is capable of translating the TORTP header into the corresponding TCP header.

When the UTCP interface receives a new packet, it uses local data and the TORTP headers to create the corresponding TCP header. The resulting packet is then injected into the TCP stack. When Daytona's TCP stack emits a new packet, a generic *tx()* method is invoked, passing only the packet and its length. We look up the corresponding UTCP circuit using the addresses and ports of the emitted TCP header, and translate the TCP header to our TORTP header and copy the TCP payload. This prepared TORTP packet is then sent to Tor, along with a reference to the appropriate circuit, and Tor sends the packet to the destination OR over the appropriate DTLS connection.

3.5 Congestion Control

The congestion control properties of the new scheme will inherit directly from those of TCP, since TCP is the protocol being used internally. While it is considered an abuse of TCP's congestion control to open multiple streams between two peers simply to send more data, in this case we are legitimately opening one stream for each circuit carrying independent data. When packets are dropped, causing congestion control to activate, it will only apply to the single stream whose packet was dropped. Congestion control variables are not shared between circuits; we discuss the possibility of using the message-oriented Stream Control Transport Protocol (SCTP), which shares congestion control information, in Section 5.2.

If a packet is dropped between two ORs communicat-

ing with multiple streams of varying bandwidth, then the drop will be randomly distributed over all circuits with a probability proportional to their volume of traffic over the link. High-bandwidth streams will see their packets dropped more often and so will back off appropriately. Multiple streams will back off in turn until the congestion is resolved. Streams such as ssh connections that send data interactively will always be allowed to have at least one packet in flight regardless of the congestion on other circuits.

Another interesting benefit of this design is that it gives Tor direct access to TCP parameters at runtime. The lack of sophistication in Tor's own congestion control mechanism is partially attributable to the lack of direct access to networking parameters at the kernel level. With the TCP stack in user space Tor's congestion control can be further tuned and optimized. In particular, end-to-end congestion control could be gained by extending our work to have each node propagate its TCP rate backwards along the circuit: each node's rate will be the minimum of TCP's desired rate and the value reported by the subsequent node. This will address congestion imbalance issues where high-bandwidth connections send traffic faster than it can be dispatched at the next node, resulting in data being buffered upon arrival. When TCP rates are propagated backwards, then the bandwidth between two ORs will be prioritized for data whose next hop has the ability to immediately send the data. Currently there is no consideration for available bandwidth further along the circuit when selecting data to send.

4 Experimental Results

In this section we perform experiments to compare the existing Tor transport layer with an implementation of our proposed TCP-over-DTLS transport. We begin by timing the new sections of code to ensure that we have not significantly increased the computational latency. Then we perform experiments on a local Tor network of routers, determining that our transport has indeed addressed the cross-circuit interference issues previously discussed.

4.1 Timing Analysis

Our UDP implementation expands the datapath of Tor by adding new methods for managing user-level TCP streams and UDP connections. We profile our modified Tor and perform static timing analysis to ensure that our new methods do not degrade the datapath unnecessarily. Experiment 2 was performed to profile our new version of Tor.

The eightieth percentile of measurements for Experiment 2 are given in Table 3. Our results indicate that no new datapath component results in a significant source of computational latency.

Experiment 2 Timing analysis of our modified TCP-over-DTLS datapath.

1: TCP-over-DTLS Tor was modified to time the duration of the aspects of the datapath:
   - injection of a new packet (DTLS decryption, preprocessing, injecting into TCP stack, possibly sending an acknowledgment),
   - emission of a new packet (header translation, DTLS encryption, sending packet),
   - the TCP timer function (increments counters and checks for work such as retransmissions and sending delayed acknowledgements), and
   - the entire datapath from reading a packet on a UDP socket, demultiplexing the result, injecting the packet, reading the stream, processing the cell, writing the result, and transmitting the generated packet.
2: The local Tor network was configured to use 50 ms of latency between connections.
3: A client connected through Tor to request a data stream.
4: Data travelled through the network for several minutes.

We have increased the datapath latency to an expected value of 250 microseconds per OR, or 1.5 milliseconds for a round trip along a circuit of length three. This is still an order of magnitude briefer than the round-trip times between ORs on a circuit (assuming geopolitically diverse circuit selection). Assuming each packet is 512 bytes (the size of a cell—a conservative estimate as our experiments have packets that carry full dataframes), we have an upper bound on throughput of 4000 cells per second or 2 MB/s. While this is a reasonable speed that will likely not form a bottleneck, Tor ORs that are willing to devote more than 2 MB/s of bandwidth may require better hardware than the Thinkpad R60 used in our experiments.

4.2 Basic Throughput

We perform Experiment 3 to compare the basic throughput and latency of our modification to Tor, the results of which are shown in Table 4. We can see that the UDP version of Tor has noticeably lower throughput. Originally it was much lower, and increasing the throughput up to this value took TCP tuning and debugging the user-level TCP stack. In particular, errors were uncovered in Daytona's congestion control implementation, and it is

Datapath Component	Duration
Injecting Packet	100 microseconds
Transmitting Packet	100 microseconds
TCP Timer	85 microseconds
Datapath	250 microseconds

Table 3: Time durations for new datapath components. The results provided are the 80th percentile measurement.

Configuration	Network Throughput	Circuit Delay	Base Delay
TCP Tor	176 ± 24.9 KB/s	1026 ± 418 ms	281 ± 12 ms
TCP-over-DTLS Tor	111 ± 10.4 KB/s	273 ± 31 ms	260 ± 1 ms

Table 4: Throughput and delay for different reordering configurations. The configuration column shows which row correspond to which version of Tor we used for our ORs in the experiment. Network throughput is the average data transfer rate we achieved in our experiment. Circuit delay is the latency of the circuit while the large bulk data transfer was occurring, whereas the base delay is the latency of the circuit taken in the absence of any other traffic.

suspected that more bugs remain to account for this disparity. While there may be slight degradation in performance when executing TCP operations in user space instead of kernel space, both implementations of TCP are based on the same Linux TCP implementation operating over in the same network conditions, so we would expect comparable throughputs as a result. With more effort to resolve outstanding bugs, or the integration of a user-level TCP stack better optimized for Tor's needs, we expect the disparity in throughputs will vanish. We discuss this further in the future work section.

More important is that the circuit delay for a second stream over the same circuit indicates that our UDP version of Tor vastly improves latency in the presence of a high-bandwidth circuit. When one stream triggers the congestion control mechanism, it does not cause the low-bandwidth client to suffer great latency as a consequence. In fact, the latency observed for TCP-over-DTLS is largely attributable to the base latency imposed on connections by our experimental setup. TCP Tor, in contrast, shows a three-and-a-half fold increase in latency when the circuit that it multiplexes with the bulk stream is burdened with traffic.

The disparity in latency for the TCP version means that information is leaked: the link between the last two nodes is witnessing bulk transfer. This can be used as a reconnaissance technique; an entry node, witnessing a bulk transfer from an client and knowing its next hop, can probe potential exit nodes with small data requests to learn congestion information. Tor rotates its circuits every ten minutes. Suppose the entry node notices a bulk transfer when it begins, and probes various ORs to determine the set of possible third ORs. It could further reduce this set by re-probing after nine minutes, after which time most of the confounding circuits would have rotated to new links.

We conclude that our TCP-over-DTLS, while currently suffering lower throughput, has successfully addressed the latency introduced by the improper use of the congestion control mechanism. We expect that once perfected, the user-level TCP stack will have nearly the same throughput as the equivalent TCP implementation in the kernel. The response latency for circuits in our improved Tor is nearly independent of throughput on existing Tor circuits travelling over the same connections; this improves Tor's usability and decreases the ability for one circuit to leak information about another circuit using the same connection through interference.

4.3 Multiplexed Circuits with Packet Dropping

Packet dropping occurs when a packet is lost while being routed through the Internet. Packet dropping, along with packet reordering, are consequences of the implementation of packet switching networks and are the prime reason for the invention of the TCP protocol. In this section, we perform an experiment to contrast the effect of packet dropping on the original version of Tor and our improved version.

We reperformed Experiment 1—using our TCP-over-DTLS implementation of Tor instead of the standard implementation—to investigate the effect of packet dropping. The results are presented in Tables 5 and 6. We reproduce our results from Tables 1 and 2 to contrast the old (TCP) and new (TCP-over-DTLS) transports.

We find that throughput is much superior for the TCP-over-DTLS version of Tor. This is likely because the

Version	Configuration	Network Throughput (KB/s)	Circuit Throughput (KB/s)	Throughput Degradation	Effective Drop Rate
TCP-over-DTLS	No dropping	284 ± 35	47.3 ± 5.8	0 %	0 %
	0.1 % (remain.)	261 ± 42	43.5 ± 7.0	8 %	0.08 %
	0.1 % (shared)	270 ± 34	45.2 ± 5.6	4 %	0.03 %
TCP	No dropping	221 ± 6.6	36.9 ± 1.1	0 %	0 %
	0.1 % (remain.)	208 ± 14	34.7 ± 2.3	6 %	0.08 %
	0.1 % (shared)	184 ± 17	30.8 ± 2.8	17 %	0.03 %

Table 5: Throughput for different dropping configurations.

Version	Configuration	Average Latency	Latency Degradation	Effective Drop Rate
TCP-over-DTLS	No dropping	428 ± 221 ms	0 %	0 %
	0.1 % (remaining)	510 ± 377 ms	20 %	0.08 %
	0.1 % (shared)	461 ± 356 ms	7 %	0.03 %
TCP	No dropping	933 ± 260 ms	0 %	0 %
	0.1 % (remaining)	983 ± 666 ms	5.4 %	0.08 %
	0.1 % (shared)	1053 ± 409 ms	12.9 %	0.03 %

Table 6: Latency for different dropping configurations.

TCP congestion control mechanism has less impact on throttling when each TCP stream is separated. One stream may back off, but the others will continue sending, which results in a greater throughput over the bottleneck connection. This is reasonable behaviour since TCP was designed for separate streams to function over the same route. If congestion is a serious problem then multiple streams will be forced to back off and find the appropriate congestion window. Importantly, the streams that send a small amount of data are much less likely to need to back off, so their small traffic will not have to compete unfairly for room inside a small congestion window intended to throttle a noisy connection. The benefits of this are clearly visible in the latency as well: cells can travel through the network considerably faster in the TCP-over-DTLS version of Tor. Despite the large confidence intervals for latency mentioned earlier, we see now that TCP-over-DTLS consistently has significantly lower latency than the original TCP Tor.

The TCP-over-DTLS version has its observed throughput and latency affected proportionally to packet drop rate. It did not matter if the drop was happening on the shared link or the remaining link, since the shared link is not a single TCP connection that multiplexes all traffic. Missing cells for different circuits no longer cause unnecessary waiting, and so the only effect on latency and throughput is the effect of actually dropping cells along circuits.

5 Alternative Approaches

There are other means to improve Tor's observed latency than the one presented in this paper. For comparison, in this section we outline two significant ones: UDP-OR, and SCTP-over-DTLS.

5.1 UDP-OR

Another similar transport mechanism for Tor has been proposed by Viecco [15] that encapsulates TCP packets from the OP and sends them over UDP until they reach the exit node. Reliability guarantees and congestion control are handled by the TCP stacks on the client and the exit nodes, and the middle nodes only forward traffic. A key design difference between UDP-OR and our proposal is that ours intended on providing backwards compatibility with the existing Tor network while Viecco's proposal requires a synchronized update of the Tor software for all users. This update may be cumbersome given that Tor has thousands of routers and an unknown number of clients estimated in the hundreds of thousands.

This strategy proposes benefits in computational complexity and network behaviour. Computationally, the middle nodes must no longer perform unnecessary operations: packet injection, stream read, stream write, packet generation, and packet emission. It also removes the responsibility of the middle node to handle retrans-

Experiment 3 Basic throughput and delay for TCP and TCP-over-DTLS versions of Tor.

1: To compare TCP and TCP-over-DTLS we run the experiment twice: one where all ORs use the original TCP version of time, and one where they all use our modified TCP-over-DTLS version of Tor.
2: A local Tor network running six routers on a local host was configured to have a latency of 50 milliseconds.
3: Two OPs are configured to connect to our local Tor network. They use distinct circuits, but each OR along both circuit is the same. The latency-OP will be used to measure the circuit's latency by sending periodic timestamp probes over Tor to a timing server. The throughput-OP will be used to measure the circuit's throughput by requesting a large bulk transfer and recording the rate at which it arrives.
4: We start the latency-OP's timestamp probes and measure the latency of the circuit. Since we have not begun the throughput-OP, we record the time as the base latency of the circuit.
5: We begin the throughput-OP's bulk transfer and measure throughput of the circuit. We continue to measure latency using the latency-OP in the presence of other traffic. The latency results that are collected are recorded separately from those of step 4.
6: Data was collected for over a minute, and each configuration was run a half dozen times to obtain confidence intervals.

missions, which means a reduction in its memory requirements. The initial endpoint of communication will be responsible for retransmitting the message if necessary. We have shown that computational latency is insignificant in Tor, so this is simply an incidental benefit.

The tangible benefit of UDP-OR is to improve the network by allowing the ORs to function more exactly like routers. When cells arrive out of order at the middle node, they will be forwarded regardless, instead of waiting in input buffers until the missing cell arrives. Moreover, by having the sender's TCP stack view both hops as a single network, we alleviate problems introduced by disparity in network performance. Currently, congestion control mechanisms are applied along each hop, meaning that an OR in the middle of two connections with different performance metrics will need to buffer data to send over the slower connection. Tor provides its own congestion control mechanism, but it does not have the sophistication of TCP's congestion control.

We require experimentation to determine if this proposal is actually beneficial. While it is clear that memory requirements for middle nodes are reduced [15], the endpoints will see increased delay for acknowledgements. We expect an equilibrium for total system memory requirements since data will be buffered for a longer time. Worse, the approach shifts memory requirements from being evenly distributed to occurring only on exit nodes—and these nodes are already burdened with extra responsibilities. Since a significant fraction of Tor nodes volunteer only to forward traffic, it is reasonable to use their memory to ease the burden of exit nodes.

Circuits with long delays will also suffer reduced throughput, and so using congestion control on as short a path as possible will optimize performance. If a packet is dropped along the circuit, the endpoint must now generate the retransmission message, possibly duplicating previous routing efforts and wasting valuable volunteered bandwidth. It may be more efficient to have nodes along a circuit return their CWND for the next hop, and have each node use the minimum of their CWND and the next hop's CWND. Each node then optimizes their sending while throttling their receiving.

5.1.1 Low-cost Privacy Attack

UDP-OR may introduce an attack that permits a hostile entry node to determine the final node in a circuit. Previously each OR could only compute TCP metrics for ORs with whom they were directly communicating. Viecco's system would have the sender's TCP stack communicate indirectly with an anonymous OR. Connection attributes, such as congestion and delay, are now known for the longer connection between the first and last nodes in a circuit. The first node can determine the RTT for traffic to the final node. It can also reliably compute the RTT for its connection to the middle node. The difference in latency reflects the RTT between the second node and the anonymous final node. An adversary can use a simple technique to estimate the RTT between the second node and every other UDP Tor node in the network [3], possibly allowing them to eliminate many ORs from the final node's anonymity set. If it can reduce the set of possible final hops, other reconnaissance techniques can be applied, such as selectively flooding each OR outside of Tor and attempting to observe an increased latency inside Tor [10]. Other TCP metrics may be amalgamated to further aid this attack: congestion window, slow-start threshold, occurrence of congestion over time, standard deviation in round-trip times, etc. The feasibility of this attack should be examined before allowing nodes who do not already know each other's identities to share a TCP conversation.

5.2 Stream Control Transmission Protocol

The Stream Control Transmission Protocol (SCTP) [13] is a message-based transport protocol. It provides sim-

ilar features to TCP: connection-oriented reliable delivery with congestion control. However, it adds the ability to automatically delimit messages instead of requiring the receiving application to manage its own delimiters. The interface is based on sending and receiving messages rather than bytes, which is appropriate for Tor's cell-based transport.

More importantly, SCTP also adds a feature well-suited to our purposes—multiple streams can be transported over the same connection. SCTP allows multiple independent ordered streams to be sent over the same socket; we can use this feature to assign each circuit a different stream. Cells from each circuit will arrive in the order they were sent, but the order cells arrive across all circuits may vary from they dispatch order. This is exactly the behaviour we want for cells from different circuits being sent between the same pair of ORs.

While SCTP is not as widely deployed as TCP, the concept of using a user-level SCTP implementation [5] inside Tor remains feasible. This suggests a SCTP-over-DTLS transport similar in design to our TCP-over-DTLS design. This means that the extra benefits of TCP-over-DTLS will also extend to SCTP-over-DTLS: backwards compatibility with the existing Tor network, a constant number of kernel-level sockets required, and a secured transport header.

What is most interesting about the potential of SCTP-over-DTLS is SCTP's congestion control mechanism. Instead of each TCP stream storing its own congestion control metrics, SCTP will share metrics and computations across all streams. An important question in the development of such a scheme is whether SCTP will act fairly towards streams that send little data when other streams invoke congestion control, and whether the sharing of congestion control metrics results in a privacy-degrading attack by leaking information.

6 Future Work

6.1 Live Experiments

The most pressing future work is to perform these experiments on live networks of geographically distributed machines running TCP-over-DTLS Tor, using computers from the PlanetLab network, or indeed on the live Tor network. Once running, we could measure latency and throughput as we have already in our experiments, comparing against results for regular Tor. Moreover, we can also compare other approaches, such as SCTP-over-DTLS and UDP-OR, using the same experiments. Note that UDP-OR could of course not be tested on the live Tor network, but it could be in a PlanetLab setup. A key metric will be the distribution of throughput and latency for high- and low-volume circuits before and after our improvements, and an analysis of the cause of the change. Additionally, once most ORs use UDP, we can determine if the reduced demand on open sockets solves the problem of socket proliferation on some operating systems.

6.2 TCP Stack Memory Management

Tor requires thousands of sockets to buffer fixed-size cells of data, but data is only buffered when it arrives out-of-order or has not been acknowledged. We envision dynamic memory management such as a shared cell pool to handle memory in Tor. Instead of repeatedly copying data cells from various buffers, each cell that enters Tor can be given a unique block of memory from the cell pool until it is no longer needed. A state indicates where this cell currently exists: input TCP buffer, input Tor buffer, in processing, output Tor buffer, output TCP buffer. This ensures that buffers are not allocated to store empty data, which reduces the overall memory requirements. Each cell also keeps track of its socket number, and its position in the linked list of cells for that socket. While each socket must still manage data such as its state and metrics for congestion control, this is insignificant as compared to the current memory requirements. This permits an arbitrary number of sockets, for all operating systems, and helps Tor's scalability if the number of ORs increases by orders of magnitude.

This approach results in the memory requirements of Tor being a function of the number of cells it must manage at any time, independent of the number of open sockets. Since the memory requirements are inextricably tied to the throughput Tor offers, the user can parameterize memory requirements in Tor's configuration just as they parameterize throughput. A client willing to denote more throughput than its associated memory requirements will have its contribution throttled as a result. If network conditions result in a surge of memory required for Tor, then it can simply stop reading from the UDP multiplexing socket. The TCP stacks that sent this unread data will assume there exists network congestion and consequently throttle their sending—precisely the behaviour we want—while minimizing leaked information about the size of our cell pool.

7 Summary

Anonymous web browsing is an important step in the development of the Internet, particularly as it grows ever more inextricable from daily life. Tor is a privacy-enhancing technology that provides Internet anonymity using volunteers to relay traffic, and uses multiple relays in series to ensure that no entity (other than the client) in the system is aware of both the source and destination of messages.

Relaying messages increases latency since traffic must travel a longer distance before it is delivered. However, the observed latency of Tor is much larger than just this effect would suggest. To improve the usability of Tor, we examined where this latency occurs, and found that it happens when data sat idly in buffers due to congestion control. Since multiple Tor circuits are multiplexed over a single TCP connection between routers, we observed cross-circuit interference due to the nature of TCP's in-order, reliable delivery and its congestion control mechanisms.

Our solution was the design and implementation of a TCP-over-DTLS transport between ORs. Each circuit was given a unique TCP connection, but the TCP packets themselves were sent over the DTLS protocol, which provides confidentiality and security to the TCP header. The TCP implementation is provided in user space, where it acts as a black box that translates between data streams and TCP/IP packets. We performed experiments on our implemented version using a local experimentation network and showed that we were successful in removing the observed cross-circuit interference and decreasing the observed latency.

Acknowledgements

We would like to thank Steven Bellovin, Vern Paxson, Urs Hengartner, S. Keshav, and the anonymous reviewiers for their helpful comments on improving this paper. We also gratefully acknowledge the financial support of The Tor Project, MITACS, and NSERC.

References

[1] Tim Dierks and Eric Rescorla. RFC 5246—The Transport Layer Security (TLS) Protocol Version 1.2. http://www.ietf.org/rfc/rfc5246.txt, August 2008.

[2] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: The Second-Generation Onion Router. *Proceedings of the 13th USENIX Security Symposium*, 2004.

[3] Krishna P. Gummadi, Stefan Saroiu, and Steven D. Gribble. King: Estimating Latency between Arbitrary Internet End Hosts. *ACM SIGCOMM Computer Communication Review*, 2002.

[4] Information Sciences Institute. RFC 793—Transmission Control Protocol. http://www.ietf.org/rfcs/rfc793.txt, September 1981.

[5] Andreas Jungmaier, Herbert Hölzlwimmer, Michael Tüxen, and Thomas Dreibholz. The SCTP library (sctplib). http://www.sctp.de/sctp-download.html, 2007. Accessed February 2009.

[6] Stephen Kent and Randall Atkinson. RFC 2401—Security Architecture for the Internet Protocol. http://www.ietf.org/rfcs/rfc2401.txt, November 1998.

[7] Marcus Leech et al. RFC 1928—SOCKS Protocol Version 5. http://www.ietf.org/rfc/rfc1928.txt, March 1996.

[8] Damon McCoy, Kevin Bauer, Dirk Grunwald, Parisa Tabriz, and Douglas Sicker. Shining Light in Dark Places: A Study of Anonymous Network Usage. University of Colorado Technical Report CU-CS-1032-07, August 2007.

[9] Nagendra Modadugu and Eric Rescorla. The Design and Implementation of Datagram TLS. *Network and Distributed System Security Symposium*, 2004.

[10] Steven J. Murdoch and George Danezis. Low-Cost Traffic Analysis of Tor. In *IEEE Symposium on Security and Privacy*, pages 183–195, 2005.

[11] Prashant Pradhan, Srikanth Kandula, Wen Xu, Anees Shaikh, and Erich Nahum. Daytona: A User-Level TCP Stack. http://nms.lcs.mit.edu/~kandula/data/daytona.pdf, 2002.

[12] Joel Reardon. Improving Tor using a TCP-over-DTLS Tunnel. Master's thesis, University of Waterloo, Waterloo, ON, September 2008.

[13] Randall Stewart, Qiaobing Xie, Ken Morneault, Chip Sharp, Hanns Juergen Schwarzbauer, Tom Taylor, Ian Rytina, Malleswar Kalla, Lixia Zhang, and Vern Paxson. RFC 2960—Stream Control Transmission Protocol. http://www.ietf.org/rfc/rfc2960.txt, October 2000.

[14] TorStatus. Tor Network Status. http://torstatus.kgprog.com/. Accessed February 2009.

[15] Camilo Viecco. UDP-OR: A Fair Onion Transport Design. http://www.petsymposium.org/2008/hotpets/udp-tor.pdf, 2008.