Weaving a Faster Tor: A Multi-Threaded Relay Architecture for Improved Throughput

Steven Engler*
Georgetown University
Washington DC, USA
steven.engler@georgetown.edu

Ian Goldberg
University of Waterloo
Waterloo, Canada
iang@uwaterloo.ca

ABSTRACT

The Tor anonymity network has millions of daily users and thousands of volunteer-run relays. Increasing the number of Tor users will enhance the privacy of not just new users, but also existing users by increasing their anonymity sets. However, growing the network further has several research and deployment challenges. One such challenge is supporting the increase in bandwidth required by additional users joining the network. While adding more Tor relays to the network would increase the total available bandwidth, it requires network architecture changes to reduce the impact of Tor's growing directory documents. In order to increase the total available network bandwidth without needing to grow Tor's directory documents, this work provides a multi-threaded relay architecture designed to improve the throughput of individual multi-core relays with available network capacity. We built an implementation of a subset of this new design on top of the standard Tor code base to demonstrate the potential throughput improvements of this architecture on both high- and low-performance hardware.

CCS CONCEPTS

• **Networks** → Network privacy and anonymity; • **Computing methodologies** → *Concurrent computing methodologies.*

KEYWORDS

Tor, performance, multi-threading

ACM Reference Format:
Steven Engler and Ian Goldberg. 2021. Weaving a Faster Tor: A Multi-Threaded Relay Architecture for Improved Throughput. In *The 16th International Conference on Availability, Reliability and Security (ARES 2021), August 17–20, 2021, Vienna, Austria.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3465481.3465745

1 INTRODUCTION

Tor, an overlay network designed for privacy, anonymity, and censorship resistance, consists of over 6500 volunteer-run relays as of March 2021 [22] with a recent estimate of 8 million active daily users [14]. While the Tor network user base is large, there is potential for it to grow significantly. The demand for web privacy tools has been demonstrated by not only the popularity of Tor, but also the popularity of the consumer VPN industry and the inclusion of private browsing modes and anti-tracking features in major browsers [18].

Growing the Tor network and its number of users is important not only for helping more people protect their online presence, but also to help existing users by increasing their anonymity set and improving their privacy. If Tor's usage is to increase, the network must have the capacity to support the additional users. Simply increasing the number of relays in the network is problematic for a number of reasons, one of which is the growing size of Tor's directory documents. While several approaches have been proposed to help the network scale, they often require network-level architectural changes. As a distributed, community-run network, these architectural changes are difficult to design, complicated to deploy, and require careful consideration to avoid partitioning the users in a way that could harm their privacy.

We propose a multi-threaded relay architecture that better utilizes existing network relays. By modifying only the internal relay architecture rather than the network architecture, the network can be improved through independent, uncoordinated upgrades to existing relays. While some relays are limited by network capacity, either from slow Internet or configuration limits set by the relay operator, other relays are not. These could range from relays operated by academic or other large institutions with high-bandwidth Internet lines, to relays running on low-performance servers such as cloud-hosted virtual private servers or small single-board computers like the Raspberry Pi. If these relays are not network-limited, they will often be CPU-constrained due to Tor's mostly single-threaded relay architecture. The multi-threaded relay architecture we describe in this work better uses the CPU resources of congested relays with available network capacity, with the goal of providing more capacity to the network to support more users. As Tor's current single-threaded relay architecture does not trivially adapt to multi-threading, this work presents a multi-threaded relay architecture, along with an implementation.

Contributions: We provide the following:

- A multi-threaded relay architecture that preserves compatibility with the existing Tor network and parallelizes the end-to-end flow of network data through the relay.
- A multi-threaded relay implementation realizing a subset of our multi-threaded architecture and a performance evaluation.

*Work done while at the University of Waterloo

ARES 2021, August 17–20, 2021, Vienna, Austria

ACM ISBN 978-1-4503-9051-4/21/08…$15.00
https://doi.org/10.1145/3465481.3465745

2 RELATED WORK

2.1 Tor

Tor is a transport-layer overlay network designed to provide anonymity at the network layer [4]. When a user application wishes to route a TCP connection through Tor, it will forward that request to a SOCKS proxy exposed by a Tor client running on the user's device. This Tor client builds multi-hop *circuits* through *relays* on the Tor network and will assign the proxy request to one of these circuits. The client can request that any hop along this circuit make the TCP connection leaving the Tor network to the destination server that was initially requested by the application, although the standard Tor implementation always requests that the final hop in the circuit make this connection. While the circuit is the multi-hop path from the Tor client to the last hop (the *exit relay*), the end-to-end path from the user's application through a circuit and to the final destination server is known as a *stream*. Tor can multiplex multiple streams through a single circuit in order to improve performance.

Clients download directory documents containing information about every relay in the network, and use these documents to construct circuits. This circuit construction uses a telescoping design that allows the Tor client at the circuit's origin to incrementally build a secure, confidential communication channel between it and each hop in the circuit. Protecting this communication is important not only for protecting the contents of the application stream, but also to prevent any intermediate relay from learning the circuit's full path during circuit creation (each relay only learns the identity of its two adjacent nodes). Since circuits typically use three hops for circuits that provide anonymity, each hop is referred to by a name given its position in the circuit: the entry relay, middle relay, and exit relay.

Tor processes communicate using a custom application-level protocol, which consists of sending *cells* across TLS-encrypted *relay connections*. Any circuits on the network that pass along this edge will be multiplexed over this connection. There are different cell types for actions such as building or destroying circuits, relaying data, padding connections, negotiating protocol versions, authenticating relays, and more. The type of cell is specified by its *cell command*, and while most of these cells are designed only to be passed between two directly connected Tor processes (whether relays or clients), Relay cells and its various sub-types are designed to pass a message along a circuit and are how a circuit's origin communicates privately with each hop along the circuit's path. These Relay cells are constructed using a layered-encryption design that allows only a specific hop to read the cell's payload. When a relay receives a Relay cell and removes the outermost layer of this encryption, the cell may be *recognized* if it is intended for this relay, or *unrecognized* if the relay should pass the cell on to the next hop in the circuit.

2.2 Tor Routing Performance

The performance of Tor's volunteer-run relays is fundamental to the health of the network, and there has been plenty of work to monitor and improve this performance [2]. For interactive applications such as web browsing, the end-to-end latencies of connections over the Tor network have a large impact on usability. Prior work has shown that relays are the main contributors to delays in Tor, largely due to relay congestion [3]. These delays vary greatly across relays and over time making them unpredictable. In order to improve the performance and usability of interactive applications, priority-based circuit scheduling was introduced as a measure to improve latency by prioritizing bursty circuits [20]. To improve the effectiveness of circuit scheduling, Jansen et al. proposed a smarter connection scheduler [8], which moves the queueing delay from the kernel's outbound queues to Tor's internal cell queues. This kernel-informed socket transport (KIST) scheduler gives Tor better scheduling control. Both of these designs are in use by Tor relays today.

To limit congestion, Tor uses a simple window-based flow control algorithm that limits the number of in-flight cells on circuits and streams. Since relays communicate at the application layer, the Tor network breaks the end-to-end congestion control principle and instead uses an entry-to-exit flow control for both circuits and streams. This flow control is in addition to the link-based TCP congestion control between relays. To examine the performance of Tor's flow control, AlSabah et al. studied the effect of smaller and dynamic window sizes on download times and time to first byte [1].

2.3 Tor Network Architecture Changes

Our work on improving the throughput of CPU-limited relays using multi-threading is largely orthogonal and complementary to other network scaling and performance improvements. The most promising network scaling approach today is Walking Onions [12], a new circuit construction design that allows the network to grow while having constant-sized directory documents and without relaxing Tor's security, correctness, or privacy properties. However, Walking Onions requires significant changes to Tor's network architecture and protocols.

A multi-threaded relay architecture enables an increase in the network capacity *without* increasing the network size or requiring any network architectural changes. For a given amount of traffic, distributing the work across multiple threads can also reduce processing and queueing delays, improving latency. Network architecture changes will be required in the future to scale beyond the improvement provided by a multi-threaded relay architecture, but these network-level scalability solutions are much more difficult to deploy. Notably, both the circuit and connection scheduling designs described in Section 2.2, which have been deployed widely on the Tor network, are relay changes and not network changes.

3 CURRENT TOR RELAY ARCHITECTURE

Before presenting a multi-threaded relay architecture, it is important to discuss Tor's current relay architecture. The multi-threaded architecture is heavily inspired by this current architecture and they share many similarities. Since we are interested in parallelizing the end-to-end processing of most network data through the relay, this section focuses on Tor's networking, its processing of different cell types, and how data is communicated between the relay's components.

As Tor is an overlay network, multiple streams can be multiplexed over a single circuit and multiple circuits can be multiplexed

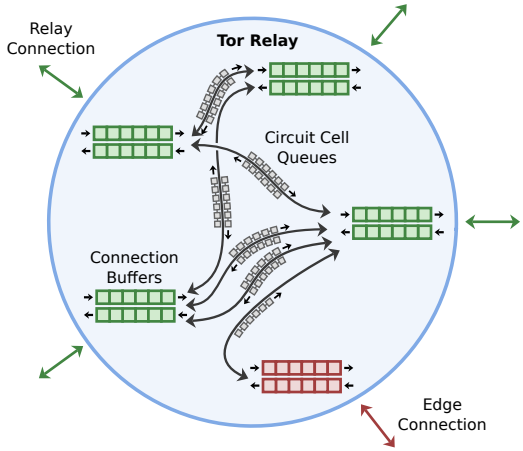


Figure 1: A basic overview of the flow of relayed data within a Tor relay.

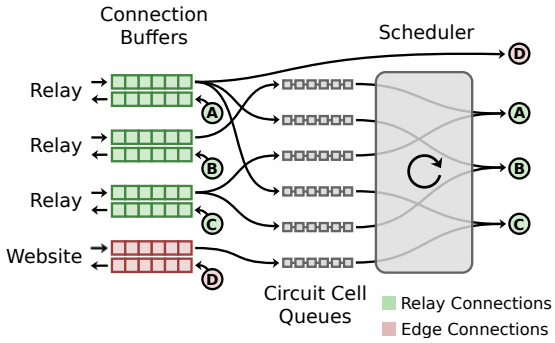


Figure 2: Tor's connection scheduler sits between the cell queues and relay connections' outbound buffers, and requires global knowledge about each connection and cell queue.

over a single relay connection. Due to this multiplexing, data received on one connection is distributed to many other connections as visualized in Figure 1. Relays must receive data on one connection, determine which circuit (and possibly which stream) it belongs to, process it, and often send it out on another connection. Within a relay, connections are intrinsically linked by circuits to other connections.

Relays classify their TCP connections into several types depending on their purpose. The important connection types for this discussion are TLS-encrypted *relay connections* (connections between Tor processes) and *edge connections* (connections entering or leaving the Tor network). For each connection Tor maintains an inbound and outbound buffer for reading/writing to the connection's socket. Circuit objects hold state for each circuit that passes through the relay (for example its forward and backward session keys) and keep references to any associated relay or edge connections. Circuit objects also keep outgoing *cell queues*, which are FIFO linked lists of cells that are to be sent on a relay connection. Relays use an event-driven design that runs a libevent-based eventloop [16] to wait for non-blocking network, timer, and signal events and run their corresponding event callbacks.

Relays must handle requests to create circuits, extend circuits, establish streams, relay data, and more. These requests cannot be handled independently as Tor's connection and circuit multiplexing requires the relay to share state among these requests. Tor stores most of this state in global lists and hash tables that are accessible everywhere within the relay. Relays use a scheduler to improve prioritization and reduce queueing delays on outbound relay connections. Tor's KIST scheduler copies cells from circuits' cell queues to their linked connections' outbound buffers, and writes the buffers to their corresponding sockets immediately within the scheduling loop as shown in Figure 2. Tor's scheduler only acts on circuits' cell queues, meaning it plays no role in cells that are added directly to a relay connection's outbound buffer, or in writing to edge connections (which do not use cell queues).

3.1 Other Relay Components

While our architecture does not consider every feature that Tor implements, there are a few other important components to consider. Relays record and limit how many bytes they read and write to the network using both token bucket-based and accounting-based bandwidth limits, some of which are global across all connections. A relay will also destroy some of its circuits and connections if its memory usage becomes too high, either due to limited resources or denial-of-service attacks [10]. The Tor network supports *onion services*, which are anonymous servers that Tor clients can connect to using a known onion service name. The client and service build their own anonymous circuits to a rendezvous relay selected by the client, and the relay links the two circuits such that cells arriving from one circuit are redirected to its paired/spliced circuit.

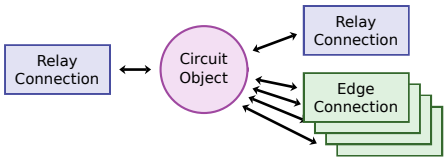
Relays use multi-threading only to move small isolated tasks outside of the main eventloop using a threadpool and work queues. Tor uses this threadpool of *CPU worker threads* for processing circuit handshakes, compressing network consensus documents for caching, and for computing the difference between cached consensus documents. While parallelizing these tasks does offer some performance improvement to the relay, the primary task of forwarding cells along circuits is not parallelized and does not scale over multiple threads. In the next section, we present a multi-threaded architecture that does parallelize this end-to-end relaying of data.

4 A MULTI-THREADED TOR RELAY ARCHITECTURE

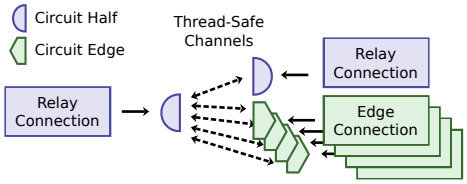
In this section we propose a multi-threaded Tor relay architecture that parallelizes the end-to-end processing of relayed data as it passes through the relay. We have three primary goals:

- Preserve compatibility with the existing live Tor network.
- Parallelize the end-to-end flow of unrecognized Relay cells throughout the relay.
- Maintain some similarity with Tor's existing relay architecture in order to increase the amount of code that can be re-used in its implementation.

The maximum throughput of a relay, assuming it is not limited by network bottlenecks or configuration options, is largely limited



(A) Tor's architecture: A single circuit object is shared by up to two relay connections and possibly many edge connections. Through the circuit object, each connection can directly access any other.



(B) Multi-threaded architecture: No connection (nor circuit half) can access their counterpart directly. Instead they can only communicate through thread-safe asynchronous channels.

FIGURE 3: Comparison of the navigability of Tor's circuit objects compared to the multi-threaded architecture's circuits. Dashed lines represent channels.

by the rate at which a relay can receive, process, and send cells. The path through the relay that these cells take is often referred to as the fast (or sometimes critical) path. This is the path from reading cells on TLS-encrypted relay connections, parsing the cells, processing them, scheduling their circuits, and writing them on a different TLS connection (or edge connection). This path is largely CPU-limited and the Tor developers have put effort into improving its performance [5, 15]. This is the path that we parallelize across threads.

If the total throughput of a relay increases, it can either handle more users at a given per-circuit throughput, or provide higher throughput per circuit for the same number of users. If the relay serves as a bottleneck for the circuit, this higher relay throughput could improve the circuit's overall throughput. Increasing the throughput of a Tor circuit has the potential to improve its overall latency as well due to Little's law [13] where a Tor circuit acts as a limited-length queue of in-flight cells in the network.

Rejected designs: We considered two alternative designs for distributing relay data processing across CPU cores. Rather than designing a new architecture for parallel end-to-end processing of cells, an alternative design could offload expensive parts of this cell processing to a work queue and threadpool. This approach would offer limited scalability since some expensive operations such as connection/circuit scheduling do not translate easily to a work queue design, and there is not a straightforward set of expensive operations that can be parallelized. Another approach might be to run multiple Tor relays on the same server (possibly one per CPU core). This approach unnecessarily grows Tor's consensus document, adds additional load to directory authorities and directory caches, limits the relay's ability to load-balance connections across CPU cores, and would require relaxing Tor's Sybil attack mitigation limit of 2 relays per IPv4 address. While these designs would likely be simpler to implement, they would not be expected to scale well across cores and have unnecessary overhead.

4.1 The Multi-threaded Relay Architecture

Our multi-threaded architecture parallelizes the end-to-end processing of relayed data by distributing its connections across additional threads with their own eventloops. These threads are responsible for reading, processing, and writing network data for each of their connections. Communication between threads uses asynchronous messages passed along thread-safe buffered channels that integrate into the eventloop. This architecture focuses on the main onion-routing tasks of a Tor relay such as creating and extending circuits, relaying data, and connection/circuit scheduling. It aims to require little shared state, remove the dependence of connection objects on one another, and provide well-defined ownership of connection and circuit objects. This leads to few required locks and safer memory management. This multi-threaded architecture is complementary to Tor's existing work queue and threadpool.

Unlike Tor's current architecture where one connection object can directly access another through a shared circuit object, this type of multi-threaded architecture requires clear separation between connections. Since connections should not share a single circuit object, circuits are instead split into two circuit halves, one for each connection. These two circuit objects have no reference to each other and can only communicate over a thread-safe channel as shown in Figure 3. This independence allows two connections to be owned by different threads while still allowing their circuits to communicate with limited and well-defined shared state (the channel) and without potentially introducing unsafe memory conditions. Channels are also used for communication between each additional thread and the main thread, which acts as a global controller for the relay.

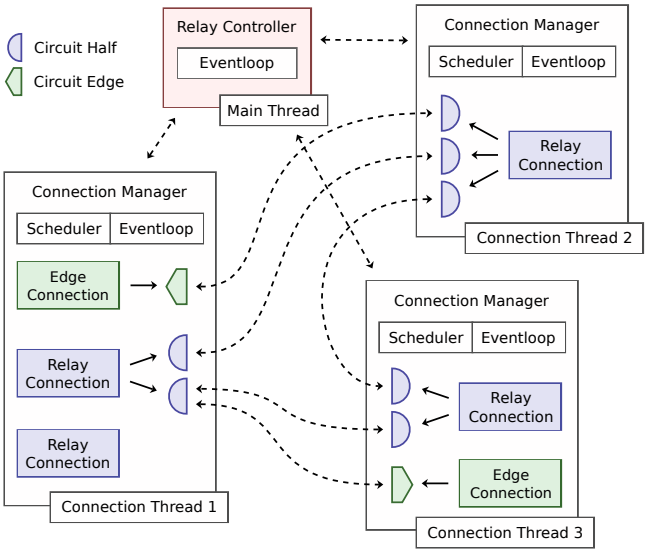
4.2 Building Blocks

The multi-threaded architecture is made up of several components, and Figure 4 presents an overview of how these components fit together.

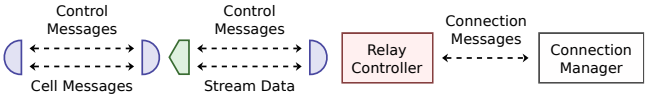
Channels. Rather than sharing state across threads, channels[1] are used to send messages and pass ownership of data between threads. Each channel should provide the ability for two threads to asynchronously send and receive messages. Channels are bi-directional, buffered, and should integrate into the eventloop such that each end of the channel is notified of messages added at the opposite end.

Circuit half and circuit edge. Circuit information is stored in circuit objects: a *circuit half* for relay connections and a *circuit edge* for edge connections. Each circuit half has its own cell queue for cells exiting the relay from its half. A circuit half can be linked by channels to at most one other circuit half, but can also be linked to many circuit edges as shown in Figure 3. For each other circuit object they are linked to, circuit objects hold ends of two different channels, one for general data/cells and the other for control

[1] Tor's implementation uses *channel objects* which were originally designed to act as a cell-handling abstraction for relay connections, but our use of the term *channel* in this paper is unrelated and refers to a means of communication between threads.



(A) Overview of the message-passing architecture using three connection threads with several connections and circuits. The relay controller runs in the main thread's eventloop, while the local connection managers each run in their own connection thread.



(B) Types of messages/data that each channel uses.

Figure 4: Overview of the message-passing architecture. Channels are represented by dashed lines.

messages (see Figure 4b). Distributing message types across two channels is useful for prioritizing control messages such as circuit destroy messages.

Local connection manager. As each connection thread must manage many connections, each thread uses a *local connection manager*. This stores each connection that it owns in a hash table (keyed by a globally unique connection identifier), and is responsible for maintaining those connections by attaching them to the local eventloop, refilling their rate-limiting token buckets, adding connection padding, etc. While these tasks do not require a global view of the relay, other tasks required by the relay, such as extending circuits or performing denial-of-service prevention, do.

Relay controller. Tasks that require a global view of the relay are instead performed by the *relay controller* running in the main thread. This relay controller holds the information it requires about all of the relay's connections and circuits, but importantly not references to the connection and circuit objects themselves as these are owned by the local connection managers. For example, this relay controller will maintain a hash table mapping the public identity digests for all connected relays to their corresponding unique connection identifiers (the same connection identifiers stored by its local connection manager). Each local connection manager has a channel between itself and the relay controller for use when it needs to perform tasks that require some global knowledge.

Eventloop. We do not define the structure of the eventloop in our architecture, but we assume a libevent-like library. Connections should be able to modify their events and enable or disable them when needed; for example, disabling a socket read event if a token bucket does not allow any more bytes to be read.

4.3 Connections and Scheduling

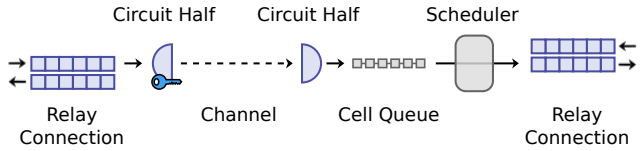
New connections are created in two cases: the relay requires a connection to an external relay or server, or one of the relay's listening sockets accepts a new connection. Both of these cases are handled by the relay controller. After opening or accepting a connection, the controller chooses a globally unique identifier for the connection. It then transfers this chosen identifier and the ownership of the connection object to one of the connection threads' local connection managers by sending a message along its channel. On receiving the message the local connection manager registers socket readable/writable events for the connection with the thread's eventloop.

Tor's scheduling loop is performance-critical and requires a global view of all connections in the relay, which is problematic in a multi-threaded architecture since the scheduler would either need to acquire locks on each connection, or use message passing to inform connections about how much data they can send. The large amount of locking would harm the relay's performance, and message passing would break some of the assumptions of Tor's primary scheduler, the KIST scheduler. Rather than using a global scheduler, each local connection manager uses its own local scheduler which processes only the connections it owns. As the connection manager can perform its scheduling operations without locking or needing to synchronize with other connection managers, this approach does not break any of the assumptions of KIST. In practice scheduling is not an operation that must provide perfect prioritization, and we do not expect that using one scheduler per thread rather than one global scheduler would significantly harm the performance of the relay or its circuits as long as connections are reasonably load-balanced across connection managers (and consequently across threads).

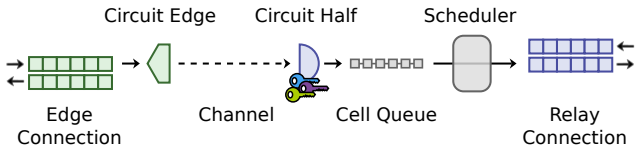
This architecture does not provide a definitive solution for load-balancing connections as this would depend strongly on the implementation and performance benchmarks. For each new connection, a trivial method of load-balancing connections would be for the relay controller to assign them to local connection managers in a round-robin manner. In practice this may perform badly as some connections will be more popular and may outlive others. Instead it might be useful for connections to occasionally report recent usage metrics (such as an exponentially weighted moving average of the bytes sent/received) back to the relay controller so that it can better load-balance new connections. The controller may also wish to occasionally move connections between local connection managers.

4.4 Extending Circuits and Relaying Data

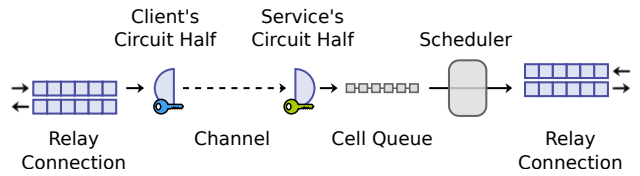
When relay connections receive cells, the connections may need to communicate with the relay controller. They do so using their local



(A) Relay cells moving forward along a circuit at a relay.



(B) Relay cells moving forward along a circuit at the circuit's origin.



(C) Relay cells (sent by the client) switch circuits at a rendezvous point.

Figure 5: Examples of the circuit objects and key locations in various circumstances. Each circuit's forward direction (away from the origin) is left-to-right.

connection manager's channel to the relay controller. For example when receiving a request to create a new circuit, the connection must query the relay controller's denial-of-service prevention subsystem by sending messages along this channel. The connection may also need to communicate with other connections. While the connection cannot communicate with other connections directly, it can do so using the relay controller as an intermediary. For example when receiving a request to extend a circuit, the connection can send a message along the same channel to inform the relay controller which connection it would like to extend a circuit to. The relay controller creates a new channel and sends each connection one end of the channel so that future cells on that circuit can be passed directly between circuit halves rather than needing to be passed through the relay controller.

From a relay's perspective, all circuits are directional and asymmetric; Relay cells travelling forward down the circuit (away from the circuit's origin) may be recognized by this relay, but Relay cells travelling backward can never be. This asymmetry means that a circuit's session keys must always be stored in the circuit half closest to the circuit's origin (see Figure 5), and that these circuit halves can operate independently of one another. For general-purpose circuits this means that only one circuit half performs cell encryption or decryption and the other will simply pass the Relay cell on. The independence between circuit halves translates naturally to circuits at rendezvous points, which link circuits together as described in Section 3.1. In this case, both circuit halves are closest to their own circuit's origin and will therefore each have their own session keys. These circuit halves can be linked with a channel just like general-purpose circuits, but instead both circuit halves will perform their own independent cell encryption or decryption.

Passing cells between circuit halves is straightforward. If one half wishes to transfer a cell to the other, it writes a reference to the cell payload and its cell command to the cell channel. The other circuit half will be notified of the new cell and can read the reference from the channel, thereby taking ownership of the cell. If it is a Relay cell and the circuit half has a session key, the circuit half may encrypt or decrypt the cell payload. The cell payload and command are then combined with the circuit identifier to form a packed cell, which can then be added to the circuit half's cell queue and later sent by the scheduler like Tor's current architecture. While a circuit half can be linked to at most one other circuit half, it may be linked to many circuit edges (one for each stream) as shown in Figure 3. When a circuit half is linked to a circuit edge, these communicate over a data channel rather than a cell channel. This channel transports raw application/stream data rather than cells. The circuit half is responsible for converting between this application data and Relay cells, and performing any required cryptography. All information that is shared between connections travel through these channels.

4.5 Bandwidth Accounting and Out-of-memory Handling

While much of the relay's global state can be handled asynchronously by the main thread's relay controller, connections may need to access some global state synchronously. For example, relays are often configured with global bandwidth limits. Each thread must make sure that they combined do not exceed any of these limits, which requires either careful synchronization of token buckets, or dividing token buckets between threads. Memory management is another component that requires a global view of the relay. Tor re-calculates its memory usage and runs its out-of-memory handler if needed for every relayed cell. This is not possible in our multi-threaded architecture, but each connection manager could periodically re-calculate the memory usage of each connection and circuit and send this information in a message to the relay controller. Each time the controller receives updated memory usage information, it can store the updated usage data and possibly choose connections or circuits to close.

4.6 Limitations

This architecture does not attempt to describe every aspect of a Tor relay. For example, we do not touch on a relay's optional directory server or onion service directory, introduction points, control connections, and more. Instead we focus on the routing tasks of a relay. That said, we do describe how some additional components fit into the architecture such as denial-of-service prevention, out-of-memory handling, and bandwidth accounting. In addition, due to the prevalence of global state in Tor and the widespread use of connection and circuit objects, implementing the architecture may require significant development effort. The Tor Project is currently in the initial stages of developing a new Tor code base in Rust using an asynchronous runtime environment (currently supporting *async-std* and *tokio*) [21], which would be an ideal time to adopt a multi-threaded architecture.

Table 1: Experiment configurations for the two servers, with one stream per circuit.

Name	Client Proxies	Entry Relays	Exit Relays	Streams per Client	Data per Stream	Total Circuits
Intel Server	150	300	300	10	10 MiB	1500
Raspberry Pi	100	300	300	6	5 MiB	600

4.7 Summary

Our multi-threaded relay architecture fulfills our three goals outlined at the start of the section:

- Preserve compatibility with the existing live Tor network — we do not require changes to any of Tor's network protocols.
- Parallelize the end-to-end flow of unrecognized Relay cells throughout the relay — we use connection threads with circuit halves and channels for communication across threads.
- Maintain some similarity with Tor's existing relay architecture in order to increase the amount of code that can be re-used in its implementation — we keep many of the same design concepts such as the scheduler, circuit cell queues, and the eventloop.

As the processing of unrecognized Relay cells does not require interaction with the main thread, we expect this architecture to scale well across threads when relaying large amounts of data. Tasks such as circuit extension, accepting incoming connections, processing consensus documents, and uploading the relay descriptor do rely on the main thread, which may become a bottleneck for the relay, but as the end-to-end processing of unrecognized Relay cells is completely independent, the throughput of existing connections and circuits should not be limited by this bottleneck. To understand how the relay scales in practice, we experiment with an implementation of key portions of our multi-threaded architecture next.

5 IMPLEMENTATION AND EVALUATION

To understand the performance of our multi-threaded architecture, we developed our relay implementation on top of the existing Tor implementation (which we refer to as *vanilla Tor*). This section describes our multi-threaded implementation and how it compares to the multi-threaded architecture, and evaluates its performance.

We created a proof-of-concept implementation of an important portion of our architecture from Section 4 and demonstrate that even this portion yields significant performance improvements (implementing the entire architecture at a production-ready level for deployment on the live network is beyond the scope of this work). This implementation demonstrates a lower bound for the performance improvement that could be obtained from implementing the entire multi-threaded architecture. We focused on parallelizing two primary components of Tor's fast path: the socket communication and TLS cryptography.

Our experiments evaluate the performance of the multi-threaded relay on two systems with very different performance characteristics. While high-performance relays will typically be running on powerful hardware, multi-threading has the potential to improve relays running on limited hardware as well. Relays with low throughput and unpredictable latency can harm the experience for users, so improving the performance of these relays is important for providing a more stable experience for users. In addition, due to their relatively low cost at around $50, these single-board computers reduce the barrier to entry for people wanting to contribute their own bandwidth to the network, or people who wish to run bridge relays for friends with censored Internet.

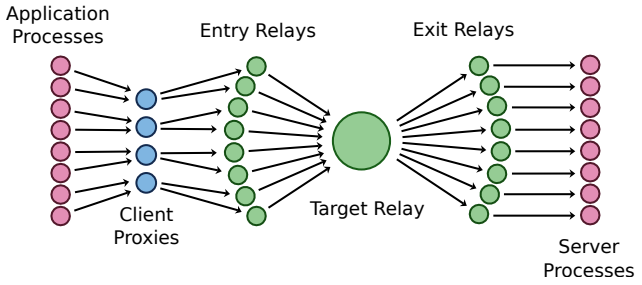


Figure 6: Overview of the experiment network and paths of circuits. Each circuit is routed through the target relay.

5.1 Multi-threaded Relay Implementation

The multi-threaded relay implementation described in this section was based on Tor 0.4.2.6 (released January 2020), and the implementation code is available at https://git-crysp.uwaterloo.ca/sengler/tor-parallel-relay-conn. The evaluation code is also available at https://git-crysp.uwaterloo.ca/sengler/relay-throughput-testing.

Tor's current relay architecture uses several design patterns that make it unsuitable for multi-threading. For example, Tor makes heavy use of global state, singletons, and circular references between connection and circuit objects. Sharing this data across threads in a safe and efficient manner is non-trivial. Due to these challenges with Tor's existing codebase, we opted (as described above) not to implement the entire multi-threaded architecture. Instead we implemented a subset of the architecture with much of the relay's cell processing remaining in the main thread. Rather than passing cells directly between connection threads, cells are passed from a connection thread to the main thread for processing, and later from the main thread to another connection thread. As described in Section 4.1, the relay's main thread starts multiple connection threads, each with its own eventloop. Rather than moving each entire connection object to a connection thread, we moved small logical pieces out of the relay connection objects and into new thread-safe objects that could be shared among threads.

5.2 Experimental Design

Our experiments were designed to evaluate the throughput of our multi-threaded relay as a middle relay (the most common type of relay on the network) by measuring the maximum sustained

Table 2: System information about the three servers used in our experiments. For Intel CPUs, core counts represent virtual cores (including hyperthreading).

Name	CPU	RAM	NIC
Intel Server	6 cores of a 16-core 2.40 GHz Intel Xeon	128 GiB	10 Gbps
Raspberry Pi	Quad-core 1.4 GHz ARM Cortex-A53	1 GiB	1 Gbps over USB 2.0
Control Server	4×16-core 2.40 GHz Intel Xeon	512 GiB	10 Gbps and 1 Gbps

throughput of the relay while under heavy CPU load. These experiments were not designed to model real-world Tor network traffic, but rather to provide a simple model for comparing specific aspects of the relay's performance. In order to saturate the relay, Tor clients (on an experimental Tor network, not the live network) build hundreds of circuits through a single target relay and simultaneously send data through these circuits to a server. Each circuit we create uses a non-exit relay as the first hop, the target relay as the middle hop, and an exit relay as the final hop as shown in Figure 6. We use the Stem library [11] to build circuits with specific paths and attach our streams directly to those circuits. Table 1 shows the various network parameters for each server.

We run all of the relay, proxy, client, and server processes on a single computer, which we will call the *control server*, except for the target relay, which runs on a separate computer. In order to test the relay performance on a variety of hardware, we run experiments with the target relay running on a modern Intel Xeon server, which we will call the *Intel server*, and also on a slower but much less expensive quad-core Raspberry Pi 3B+ single-board computer[2] lacking hardware-accelerated AES support, which we will call the *Raspberry Pi server*. Table 2 shows the configuration of each system. When running on the Intel server, we assign the target relay process a NUMA policy with 6 virtual CPU cores (3 physical cores and their paired hyperthreading cores) and memory on the same NUMA node.

We run the target relay in a few different configurations for performance comparisons. As a baseline we use the standard Tor implementation (vanilla Tor) with a small patch to log its throughput. We also use our multi-threaded relay implementation with the same patch and run it with varying numbers of relay connection threads. Each client and relay in these experiments is based on Tor 0.4.2.6. We also disabled the directory cache functionality on the target relays since our experimental Tor network used a consensus voting interval of 40 seconds compared to 1 hour on the real Tor network to speed up bootstrapping.

Once the circuits have been built on all clients, we start application processes, which connect to the SOCKS ports on our clients. Each of these application streams is assigned to a different circuit so that there is at most one stream per circuit. These streams connect through the Tor network to a server, which forks and handles each stream in its own process. Once all of the application streams have been attached to circuits, all client processes begin simultaneously sending data through the Tor network to the server. All of these circuits pass through the target relay, and since the control server uses significantly more CPU cores than the target relay to process the same amount of network data, the target relay becomes the bottleneck limiting the application stream throughput. As the server processes receive data, they record the time and number of bytes read from the socket. Eventually all server processes receive their data and the experiment ends.

[2]The Raspberry Pi's Ethernet adapter is attached over USB 2.0, limiting the effective bi-directional network bandwidth to ~150 Mbps.

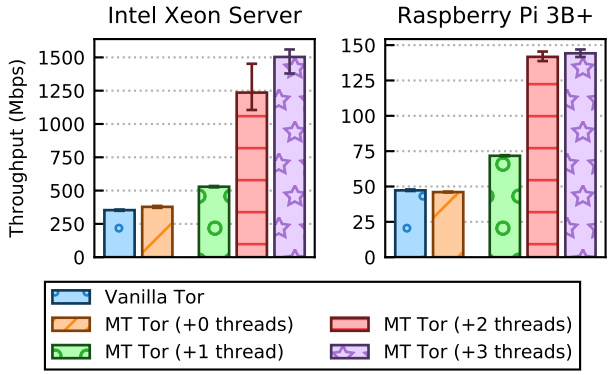
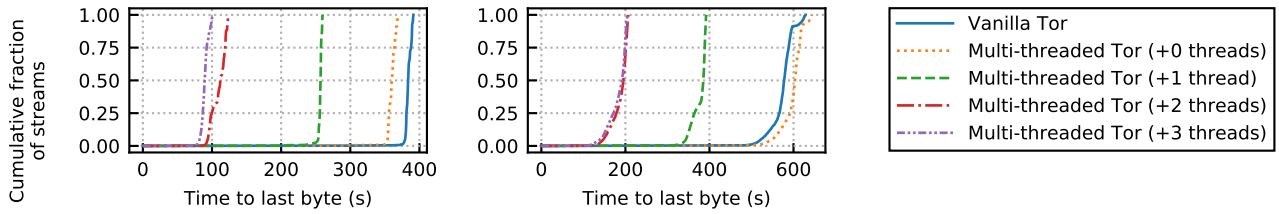


Figure 7: Maximum sustained throughput over 30 seconds of an original Tor relay and a multi-threaded Tor relay with varying numbers of connection threads. Error bars show the minimum and maximum values measured over 10 repetitions. All relay versions use the jemalloc memory allocator.

5.3 Results

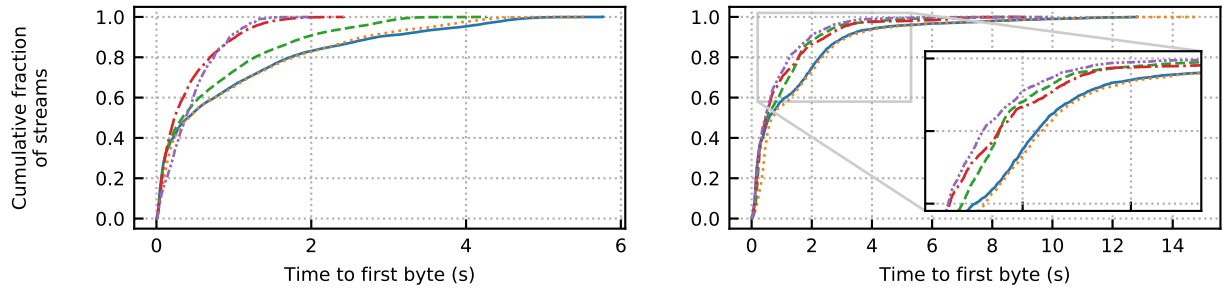
Following the experimental setup in Section 5.2, we repeated each experiment configuration 10 times. Tor's current relay implementation is identified by "vanilla Tor", and the multi-threaded relay implementation is shortened to "MT Tor". All of the results in this section ignore Tor's CPU-worker threads (see Section 3.1), which were almost completely unused and have an insignificant effect on the CPU performance and throughput of the relay in these experiments. For results with the multi-threaded relay implementation where there are no connection threads, all code is run in the main thread much like vanilla Tor. All results are shown using the jemalloc memory allocator due to its better multi-threaded performance compared to the GNU glibc allocator, with no detrimental effect on single-threaded performance.

5.3.1 Throughput. Each experiment logs the number bytes sent and received on relay connections every 500 ms. From this data we find the maximum sustained throughput over a 30 second period (the maximum of a 30 second moving average), discarding the first 30 seconds of throughput data. As shown in Figure 7, we see a significant throughput improvement with our multi-threaded relay when using several threads. When running with three connection threads, the maximum sustained throughput is about 4.3 times greater than vanilla Tor on the Intel server, and about 3.1 times greater on the Raspberry Pi server. The maximum throughput of the multi-threaded relay on the Raspberry Pi is very close to the limit of its USB-attached network adapter (about 150 Mbps).



(A) Intel server (B) Raspberry Pi

Figure 8: The time to last byte (upload time) for each stream combined for all 10 repetitions.



(A) Intel server (B) Raspberry Pi

Figure 9: The time to first byte for each stream combined for all 10 repetitions.

When assessing the performance scaling of our relay implementation with varying numbers of threads, the single-threaded performance is not directly comparable to the multi-threaded performance since the single-threaded case is handled differently. When running our relay implementation with only a single thread (no connection threads), the connection processing code is run in the main thread and shares the CPU core's capacity with other processing performed by the relay in the same thread. This is useful for comparing this special case to vanilla Tor. When running with a single thread (no connection threads), the multi-threaded relay implementation performs very similar to vanilla Tor even with the multi-threaded relay's additional overhead from message passing.

5.3.2 Stream Performance. The previous results have looked at the performance of the relay as a whole, but the performance of the individual application streams is also important as it influences the end user's experience. We measure the streams' upload times (time to last byte) and their initial upload latencies (time to first byte), and show them in Figures 8 and 9. When using no connection threads, the multi-threaded relay performed similar to vanilla Tor, showing that the multi-threaded relay does not have significantly worse latency or throughput compared to vanilla Tor when running only in the main thread.

As expected since all streams are effectively the same, all of the streams complete around the same time, demonstrating that our multi-threaded relay does not seem to negatively affect the relay's circuit prioritization mechanisms in our experiments. When scaling the multi-threaded relay to additional connection threads, streams completed much quicker due to the higher relay bandwidth, which corresponds to much faster data transfer rates for clients. Since all streams are started at the same time, the relay quickly becomes saturated, and the time to first byte of each stream corresponds to the latency during this time. Figure 9 shows that the multi-threaded relays had lower initial latencies than vanilla Tor.

5.4 Discussion

Our relay implementation successfully used multiple CPU cores to improve the relay's throughput. The maximum sustained throughput tripled on the Raspberry Pi and quadrupled on the Intel server when running with three connection threads. The initial latency measurements also improved when using our implementation. With a throughput of nearly 150 Mbps, the Raspberry Pi running the multi-threaded implementation becomes a much more viable system for running an inexpensive Tor relay.

The multi-threaded relay implementation parallelizes only the relay connection networking component of the architecture described in Section 4, but already demonstrates a significant performance improvement of a multi-threaded relay on both limited and high-performance hardware. It is likely that implementing more components of the architecture would lead to better scaling over more CPU cores, as our implementation does not fully realize the fully parallel end-to-end processing of cells from the multi-threaded architecture. One of the ways in which our implementation differs from our architecture is that cells always pass through the main thread as described in Section 5.1 instead of moving directly between connection threads. This requires that our implementation queue more cells than both vanilla Tor and what our architecture requires, and means that our implementation has a higher memory usage, but this is a limitation of our proof-of-concept implementation only and not the architecture itself.

The experiments performed in this section do not attempt to model the real-world Tor network, and it is possible that in practice the relay may not scale as well as it does in these experiments. While using more realistic models might provide more realistic results, our main objective is to provide a multi-threaded relay architecture, and demonstrate its feasibility.

6 FUTURE WORK AND CONCLUSION

We do not have a good way to measure what fraction of relays are CPU-limited from publicly available data, and Tor's bandwidth measurement infrastructure is designed to prevent relays from running at full CPU usage in order to leave CPU headroom for bursty or varying traffic. This makes it difficult to distinguish between CPU bottlenecks, a lack of user traffic on the network, or side effects of Tor's bandwidth measurement infrastructure. In the future, Tor relays may report CPU metrics that would provide real-world insights into relay performance [6].

While not all relays today are CPU-limited, many more might become so as the demand on the network grows. Providing better CPU utilization using multi-threading has the potential to improve both high- and low-performance Tor relays. Congested relays have been shown to cause a large negative impact on stream latencies and throughput [3, 19], so reducing the opportunity for congestion to occur will help the network better handle more traffic and users. Multi-core CPUs are commonplace today and while clock speeds of desktop and server x86 CPUs see marginal yearly gains, core counts have grown significantly in recent years. The network should be capable of fully utilizing the resources donated by the community.

6.1 Security and Privacy

Our multi-threaded architecture does not introduce any new attacks on users or the network,[3] and we note that supporting more users and growing the network can improve privacy for existing users by increasing their anonymity set.

One important consideration, however, is the effect on attacks that rely on distinguishable relay throughputs. For example, an adversary can probabilistically identify the relay that bottlenecks a circuit's throughput by correlating the circuit's throughput with individual relay throughput measurements taken using active probing [17]. This attack depends on a wide range of relays with different throughput characteristics, and the faster throughput enabled by multi-threading may widen the gap between the slowest and fastest relays, making it easier for attackers to distinguish relays by their throughput. Furthermore, attacks that rely on circuit latencies, in particular the round-trip time between the client and the exit [7], may improve if relays' processing and queueing delays become more consistent. While the effectiveness of these attacks may increase as the network performance improves, these attacks should not discourage performance and scaling improvements. Defences should affirmatively address these attacks rather than rely on limitations of the current deployed network.

[3]The security of our architecture is not to be conflated with the security of our proof-of-concept implementation that was built for the experiments in Section 5, and not intended for deployment on the live network.

6.2 Future Work

When scaling our implementation to several CPU cores, the relay's main thread becomes the bottleneck limiting the relay's throughput. It is likely that the entire multi-threaded architecture would scale much better, since it performs the end-to-end processing of Relay cells completely outside of the main thread, unlike our implementation where much of the cell processing remains in the main thread. Implementing the remainder of the architecture and performing throughput experiments would better show the scaling of the architecture across CPU cores. In addition, our architecture attempts to remain similar to Tor's existing relay architecture, but redesigning the architecture to make use of user-level threads and a blocking networking API may significantly reduce the complexity of the architecture and allow for a simpler relay implementation. Finally, the experiments in Section 5.2 studied only the relay performance and not the performance of the network. It would be useful to simulate a much larger and more realistic network to understand the effects on the network as a whole and its individual users. However, we note that Tor network simulators such as Shadow [9] do not capture the effects of CPU load, and so how to best perform this simulation is an open question.

6.3 Conclusion

With the objective of scaling the Tor network through internal relay architecture improvements, this work presented a multi-threaded relay architecture designed to parallelize the processing of relayed data. By implementing a subset of this architecture and examining its throughput and CPU performance in an experimental environment designed to flood the relay with traffic, we showed that this design can significantly improve a relay's throughput when CPU constrained. As we do not change any aspects of the network's architecture or protocols, relays using our multi-threaded architecture could be easily deployed by relay operators. CPU-limited relays with excess bandwidth capacity would be able to contribute more bandwidth to the network when using our multi-threaded architecture.

ACKNOWLEDGMENTS

This work benefited from the use of the CrySP RIPPLE Facility at the University of Waterloo. We gratefully acknowledge the Royal Bank of Canada and NSERC grant CRDPJ-534381 for funding this work. This research was undertaken, in part, thanks to funding from the Canada Research Chairs program.

REFERENCES

[1] Mashael AlSabah, Kevin Bauer, Ian Goldberg, Dirk Grunwald, Damon McCoy, Stefan Savage, and Geoffrey M. Voelker. 2011. DefenestraTor: Throwing out Windows in Tor. In *11th Privacy Enhancing Technologies Symposium*. Springer, 134–154.

[2] Mashael AlSabah and Ian Goldberg. 2016. Performance and Security Improvements for Tor: A Survey. *ACM Computing Surveys (CSUR)* 49, 2 (2016), 1–36.

[3] Prithula Dhungel, Moritz Steiner, Ivinko Rimac, Volker Hilt, and Keith W. Ross. 2010. Waiting for Anonymity: Understanding Delays in the Tor Overlay. In *10th IEEE Conference on Peer-to-Peer Computing (P2P)*. IEEE, 1–4.

[4] Roger Dingledine, Nick Mathewson, and Paul Syverson. 2004. Tor: The Second-Generation Onion Router. *13th USENIX Security Symposium* (2004).

[5] David Goulet. 2018. cmux: Refactor, test and improve performance of the circuitmux subsystem. https://gitlab.torproject.org/legacy/trac/-/issues/25328.

[6] David Goulet and Mike Perry. 2020. Make Relays Report When They Are Overloaded. https://gitweb.torproject.org/torspec.git/tree/proposals/110-avoid-infinite-circuits.txt.
[7] Nicholas Hopper, Eugene Y. Vasserman, and Eric Chan-Tin. 2010. How Much Anonymity does Network Latency Leak? *ACM Transactions on Information and System Security (TISSEC)* 13, 2 (2010), 1–28.
[8] Rob Jansen, John Geddes, Chris Wacek, Micah Sherr, and Paul Syverson. 2014. Never Been KIST: Tor's Congestion Management Blossoms with Kernel-Informed Socket Transport. In *23rd USENIX Security Symposium*. 127–142.
[9] Rob Jansen and Nicholas Hopper. 2012. Shadow: Running Tor in a Box for Accurate and Efficient Experimentation. In *19th Symposium on Network and Distributed System Security (NDSS)*. Internet Society.
[10] Rob Jansen, Florian Tschorsch, Aaron Johnson, and Björn Scheuermann. 2014. The Sniper Attack: Anonymously Deanonymizing and Disabling the Tor Network. *Network and Distributed System Security Symposium (NDSS)* (2014).
[11] Damian Johnson. 2020. Stem. https://stem.torproject.org/.
[12] Chelsea Komlo, Nick Mathewson, and Ian Goldberg. 2020. Walking Onions: Scaling Anonymity Networks while Protecting Users. *29th USENIX Security Symposium* (2020).
[13] John D. C. Little. 1961. A Proof for the Queuing Formula: L=λW. *Operations Research* 9, 3 (1961), 383–387.
[14] Akshaya Mani, T. Wilson-Brown, Rob Jansen, Aaron Johnson, and Micah Sherr. 2018. Understanding Tor Usage with Privacy-Preserving Measurement. In *Internet Measurement Conference*. 175–187.
[15] Nick Mathewson. 2018. See if we can allocate less for HMAC in Tor relays. https://gitlab.torproject.org/legacy/trac/-/issues/25007.
[16] Nick Mathewson, Azat Khuzhin, and Niels Provos. 2017. libevent – an event notification library. https://libevent.org/.
[17] Prateek Mittal, Ahmed Khurshid, Joshua Juen, Matthew Caesar, and Nikita Borisov. 2011. Stealthy Traffic Analysis of Low-Latency Anonymous Communication Using Throughput Fingerprinting. In *18th ACM Conference on Computer and Communications Security*. 215–226.
[18] Mozilla. 2020. Firefox privacy, by the product. https://www.mozilla.org/en-CA/firefox/privacy/products/.
[19] Joel Reardon and Ian Goldberg. 2009. Improving Tor using a TCP-over-DTLS Tunnel. In *USENIX Security Symposium*. 119–134.
[20] Can Tang and Ian Goldberg. 2010. An Improved Algorithm for Tor Circuit Scheduling. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*. 329–339.
[21] The Tor Project. 2021. Arti tor-rtcompat library (commit 541883f3). https://gitlab.torproject.org/tpo/core/arti/-/blob/541883f3dfb3de25104ade77e3c623e6a6c1a8f4/tor-rtcompat/src/lib.rs.
[22] The Tor Project. 2021. Tor Metrics — Servers. https://metrics.torproject.org/networksize.html?start=2021-01-01&end=2021-03-01.