Privacy Preserving Performance Enhancements for Anonymous Communication Networks

A DISSERTATION
SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL
OF THE UNIVERSITY OF MINNESOTA
BY

Robert G. Jansen

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
Doctor of Philosophy

Nicholas J. Hopper

October, 2012

Acknowledgements

I would like to extend much gratitude to my advisor, Nick Hopper, not only for his support of and contribution to the ideas within this dissertation, but also for his unending patience and exuberance during my development as a researcher. His qualities as an advisor are unparalleled.

I thank my other committee members – Yongdae Kim, Andrew Odlyzko, Paul Syverson, and Jon Weissman – for their comments and support of this dissertation.

I thank my collaborators – Kevin Bauer, Roger Dingledine, Nick Hopper, Aaron Johnson, Yongdae Kim, and Paul Syverson – all of whom helped shape and enhance this work. I additionally thank everyone with which I had fruitful discussions related to this work during my time in graduate school, especially: Eric Chan-Tin; Ian Goldberg; John Geddes; Denis Foo Kune; Zi Lin; Harsha Madhyastha; Prateek Mittal; Abedelaziz Mohaisen; Max Schuchard; Micah Sherr; Chris Wacek; and Eugene Vasserman.

I most extensively thank Leiah, my best friend and partner, for her patience and emotional support throughout graduate school. Her consistently positive attitude continuously bolstered my perserverence against the numerous challenges along the way.

Finally, I thank my family for their understanding and support of my pursuits.

Dedication

for $\mathcal{L}$eiah

Abstract

An anonymous communication system hides the fact that two parties are communicating, and as a result, drastically improves the online privacy of those using it. Tor is the most popular anonymous communication system deployed, but its popularity has illuminated problems with its design that have made it unbearably slow for many users who would otherwise benefit from its protections. These performance problems have been recognized, but there has been little work on designing and properly evaluating practical solutions that improve performance while also preserving privacy.

We initiate an exploration into Tor's system design and the quality of the communication it provides. First, we design and develop a simulation tool, called Shadow, that allows us to experiment with the Tor software in a safe but realistic and controllable manner. We then give a precise model of the Tor network, the backbone networks upon which it operates, and the user agents operating within it. We show that by combining our model with Shadow, our experimentation environment is capable of producing network interactions and performance qualities indicative of real systems.

We then investigate performance enhancements in three major areas of Tor's design. We explore Tor's *utilization of resources* by evaluating both existing and new circuit scheduling techniques, and show the extent to which scheduling can be used to prioritize traffic in order to improve desirable quality metrics. We then design and evaluate algorithms focused on *reducing network load* by throttling agents that consume an unfair share of network resources. Finally, in an effort to supplement Tor's volunteered resources, we design and analyze two schemes that *increase network capacity* by providing incentives to those contributing resources to the system.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Internet communication may be exemplified by a postal service: messages are sent to third parties for transport through a network of service providers and eventual delivery to the intended recipient. The message contains both a delivery (destination) address and a return (source) address. These addresses allow the third party providers to route the message toward the recipient, and allow the recipient to reply to the sender. The sender and receiver addresses that are required for proper routing leak a significant amount of information to those providing the transport service, even if the message itself is sealed. Any service provider on the path between the sender and receiver may use the source and destination addresses to determine who is talking to whom.

Internet communication occurs similarly: Internet service providers (ISPs) learn the source and destination addresses as they route data packets over the large integrated network of cables and wires that connect our communication devices. Despite confidentiality of the packet data, ISPs may infer packet contents by tracking packet sizes and the frequency at which they are sent. ISPs use this information to build profiles of senders' behaviors over time. Even if ISPs originally had good intentions when building these profiles, they may later be legally required or politically coerced to share what they know [1, 2], and may even take action against the source associated with a given type of profile [3]. These actions generally threaten Internet privacy, freedom of speech, and network neutrality.

Privacy encroachments similarly occur at many organizations around the world, most of whom have a significant interest in tracking information into and out of their networks. Institutions may be concerned that employees will utilize their networks in order to illegally access copyrighted content, or that employees spend too much time using the Internet for personal rather than work-related activities. Therefore, many employers track their employees' communication patterns and build profiles of their habits. Corporations that collect customer records or store customer data, such as cloud storage providers and location-based service providers, may be required to release that data to authorities [4]. In all of these cases, it is possible that individuals may be harassed when their information is sold, shared, or stolen.

Even browsing the web represents a significant threat to privacy. For example, Internet search websites build, collect, and store profiles about who searches for what, and when and where those searches were conducted. The profiles are used to enhance online services and target advertisements to those the searcher is most likely to prefer. Unfortunately, sensitive search information is too often leaked to or shared with the public [5, 6], and may be used to uniquely identify the searcher [7].

Government surveillance also threatens the rights of individuals. Governments surveil the communication of their citizens and block access to content that may harm their political image. They may block websites or media and track what their citizens are posting online. Some governments, e.g. those in the Arab world, have been known to threaten those who oppose their reign with violence [8]. This type of repression and censorship severely cripples what many believe is a basic right – the right to speak freely.

The importance of private communication extends beyond freedom of speech and protection against the invasive practices outlined above [9]. For example, intelligence and law enforcement agents depend on the ability to communication privately in order to conceal their operations against criminal or terrorist regimes. Businesses depend on private communication to prevent industrial espionage, competitive intelligence gathering, and the leakage of trade secrets. Journalists and political activists need private communication in order to safely plan, organize, and promote awareness of corruption, questionable tactics, or criminal behaviors. The wide and diverse set of users and groups with strong desires to maintain their privacy online motivates the design of technologies that enable safe electronic communication.

There has been a significant amount of research into *anonymous communication* in the fight for privacy and freedom of speech. A major goal of most anonymous communication research is to build, understand, and improve protocols for communicating over existing Internet transports while preventing third parties from determining *who is talking to whom* and who is accessing which networks or services.

Onion routing [10], particularly as deployed in Tor [11, 12], is the most widely used and extensively studied approach to anonymous communication. As shown in Figure 1.1, Tor is an overlay network of relays that allows its users to connect to generic Internet services. To achieve anonymity while communicating, Tor *clients* first build virtual *circuits* through a small set of available *relays* selected from Tor's *directory service*. Clients package data into 512-byte packets called *cells* and encrypt them once for each relay they selected for their circuit. Each relay decrypts its "layer" of each multiply-encrypted cell before forwarding it to the next relay in the circuit. The last

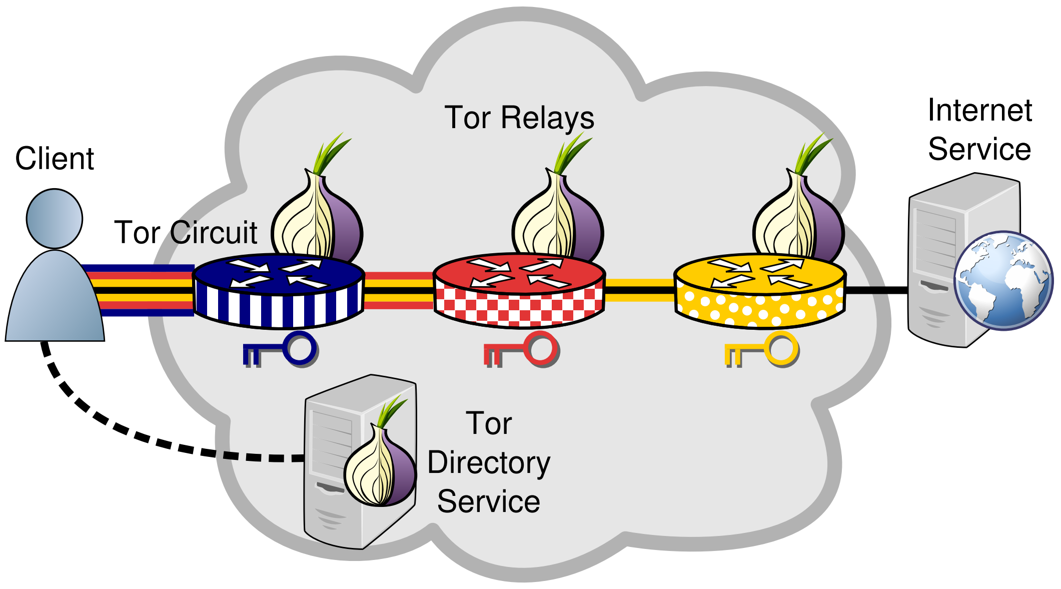


Figure 1.1: Tor network overview. Clients download relay information from the directory service before building a cryptographic virtual circuit through the chosen relays. Connections to the Internet are then tunneled through the circuit, while relays decrypt and encrypt outgoing and incoming messages, respectively. No single relay can link the client to its chosen destination.

relay forwards the then-unencrypted data to the user-specified destination, which may be a service outside the Tor network. Each relay can determine only its predecessor and successor in the path from source to destination, preventing any single relay from linking the sender and receiver. Clients also choose their first relay from a small set of *entry guards* [13, 14] in order to help defend against passive logging attacks [15], and while traffic analysis is still possible [16, 17, 18, 19, 20, 21, 22, 23], it is slightly complicated since each relay simultaneously services multiple circuits.

1.1 Performance Problems

The Tor network suffers from performance problems [24], primarily because there is too much *demand* for the existing network *capacity*. In Tor's current resource model, its popularity harms its usability and performance, and may therefore have a significantly negative impact on its users' anonymity [25, 26].

1.1.1 Demand for Bandwidth

Tor services hundreds of thousands of clients [27], some with high bandwidth demands: the most recent studies of Tor exit traffic showed that file sharing connections accounted for roughly forty percent of the data Tor transferred in 2008 [28], and fifty-two percent in 2010 [29]. The sheer number of clients and the data they want to transfer through Tor places an extreme amount of load on the network. Further, the aggregate bandwidth costs of sending data securely through multiple relays are significantly higher than direct communication: for every byte transferred between the client and the destination server, each of three relays must also transfer it downstream, and then back upstream. Fundamentally, a circuit's sustained throughput cannot be higher than the rate of the lowest upstream or downstream link of which it is composed. Performance problems are created as a result, and exacerbated when such a high demand for bandwidth is combined with relays providing relatively low network capacity.

1.1.2 Network Capacity

Tor's network consists of a few thousand relays that are run by geographically diverse volunteers [30] who altruistically contribute bandwidth and computational resources to the network. As a result, Tor is usable even by those unable or unwilling to contribute because they, e.g., have slow connections or are behind restrictive firewalls. Unfortunately, network capacity is limited to these altruistic contributions and has increased sublinearly to Tor's popularity.

A relay's utility to Tor is dependent on both the bandwidth capacity of its host network and the bandwidth restrictions imposed by its operator. Although bandwidth donations vary widely, the majority of relays offer (and in some cases can only provide) less than 100 KiB/s. These low bandwidth relays actually become bottlenecks when chosen for a circuit. Circuit bottlenecks are also created when high capacity relays become severely overloaded due to demand. Relay bottlenecks increase network-wide congestion and impair client performance, deterring users as they attempt to interactively browse the web.

Tor's capacity problems also make it vulnerable to a simple denial of service (DoS) attack on the network: with nothing but a moderate number of bulk clients, an adversary can intentionally and significantly degrade the performance of the entire Tor network for most users. This is a malicious attack as opposed to an opportunistic use of resources without regard for the impact on legitimate users, and could be used by censors [31] to discourage use of Tor. Bulk traffic effectively averts potential users from Tor at the network's current capacity, decreasing both Tor's client diversity and anonymity [25, 32].

1.2 Enhancing Performance While Preserving Privacy

This dissertation explores the following thesis:

> *Performance in existing anonymous communication systems may be enhanced, while preserving privacy, by improving the utilization of resources, reducing the network load, or increasing the network capacity.*

We will explore performance in terms of measurable network characteristics and how they change. We consider enhancing performance to mean improving the desired quality of these characteristics. Our exploration will mainly consider expected file download times as a metric for client performance: we will seek to provide lower latency and higher throughput for Tor clients to reduce the time it takes to download files through the Tor network. We will also consider overall network utilization, as we argue that network performance is enhanced if the network supports more load without reducing expected client performance.

Performance enhancements that preserve privacy do not leak any information beyond what is already leaked by existing Tor protocols. Toward this goal, our work explores algorithmic designs that do not require input from external network nodes. In other words, a node should only have access to local and previously available information when deciding how to process an algorithm or protocol. We will quantify the extent to which privacy may be preserved while using our enhancements.

1.3 Contributions and Outline

In this dissertation, we develop Tor network experimentation and modeling tools and use them to explore privacy preserving network enhancements that improve our understanding of Tor's performance problems while attempting to alleviate them. The remainder of this chapter outlines our primary contributions in this regard.

Network Experimentation Framework (Chapter 3)

New Tor design proposals and attacks on the system are challenging to test in the live Tor network because of deployment issues and the risk of invading users' privacy, while alternative Tor experimentation techniques are limited in scale, are inaccurate, or create results that are difficult to reproduce or verify. In Chapter 3 we design and implement Shadow, an architecture for efficiently running accurate Tor experiments on a single machine. We validate Shadow's accuracy with a private Tor deployment on PlanetLab and a comparison to live network performance statistics. Our software runs without root privileges, is open source, and is publicly available for download.

Tor Network Model (Chapter 4)

Live Tor network experiments are difficult due to Tor's distributed nature and the privacy requirements of its client base. Alternative experimentation approaches, such as simulation and emulation, must make choices about how to model various aspects of the Internet and Tor that are not possible or not desirable to duplicate or implement directly. In Chapter 4 we methodically model the Tor network by exploring and justifying every modeling choice required to produce accurate Tor experimentation environments. We validate our model using two state-of-the-art Tor experimentation tools and measurements from the live Tor network. We find that our model enables experiments that characterize Tor's load and performance with reasonable accuracy.

Improving Resource Utilization (Chapter 5)

One approach to improving Tor's performance is to better utilize the available network resources, particularly through scheduling. In Chapter 5 we first evaluate past and current Tor scheduling approaches [33] in our network model to understand their effects on performance. We then explore the extent to which scheduling can fundamentally benefit performance by considering schedulers that have ideal information: those that can perfectly categorize data by traffic type and prioritize it accordingly. To this end, we design and develop two schedulers based on the proportional differentiation architecture [34], implement them in Tor, and evaluate them with both single-circuit and full-network experiments to better understand how they might enhance Tor client performance.

Reducing Load (Chapter 6)

The scheduling approach attempts to reorder and prioritize packets to better utilize the available bandwidth for specific traffic classes, but does not reduce bottlenecks introduced by the massive amount of bulk traffic plaguing Tor [28]. Further, scheduling does not directly address or provide adequate defense against performance degradation attacks similar to the problems created by bulk traffic clients.

Equipped with mechanisms from communication networks, in Chapter 6 we design and implement three Tor-specific algorithms that throttle bulk transfers to reduce network congestion and increase network responsiveness. Unlike existing techniques, our algorithms adapt to network dynamics using only information local to a relay. We experiment with full-network deployments of our algorithms under a range of light to heavy network loads. We find that throttling results in significant improvements to web client performance while mitigating the negative effects of bulk transfers. We also analyze how throttling affects anonymity and compare the security of our algorithms under adversarial attack. We find that throttling reduces information leakage compared to unthrottled Tor while improving anonymity against realistic adversaries.

Increasing Capacity (Chapters 7 and 8)

A significant problem faced by the current Tor system is how to recruit new relays to increase network capacity, support expansion, and ease the load suffered by current relays. We explore two designs that attempt to solve this problem by offering performance incentives to those who contribute resources to the network.

In Chapter 7 we explore BRAIDS, a set of practical mechanisms that encourages users to run Tor relays, allowing them to earn credits redeemable for improved performance of both interactive and non-interactive Tor traffic. These performance incentives will allow Tor to support increasing resource demands with almost no loss in anonymity: BRAIDS is robust to well-known attacks. We evaluate BRAIDS and show that it allows relays to achieve lower latency than non-relays for interactive traffic, and higher bandwidth utilization for non-interactive traffic.

With similar motivations, Chapter 8 explores LIRA, a lightweight scheme that creates performance incentives for users to contribute bandwidth resources to the Tor network. LIRA uses a novel cryptographic lottery: winners may be guessed with tunable probability by any user or bought in exchange for resource contributions. The traffic of those winning the lottery is prioritized through Tor. The uncertainty of whether a buyer or a guesser is getting priority improves the anonymity of those purchasing winners, while the performance incentives encourage contribution. LIRA is more lightweight than prior reward schemes that pay for service and provides better anonymity than schemes that simply give priority to traffic originating from fast relays. We analyze LIRA's efficiency, anonymity, and incentives, present a prototype implementation, and evaluate it with experiments that show it indeed improves performance for those servicing the network.

Chapter 2

Background

2.1 Introduction

This chapter discusses Tor's internal architecture to facilitate an understanding of how internal processes affect client traffic flowing through a Tor relay. Please see Figure 2.1 for a schematic diagram.

2.2 Multiplexed Connections

All relays in Tor communicate using pairwise TCP *connections*, i.e. each relay forms a single TCP connection to each other relay with which it communicates. Since a pair of relays may be communicating data for several *circuits* at once, all circuits between the pair are multiplexed over their single TCP connection. Each circuit may carry traffic for multiple services or *streams* that a user may be accessing. TCP offers reliability, in-order delivery of packets between relays, and potentially unfair kernel-level congestion control when multiplexing connections [35]. The distinction between and interaction of connections, circuits, and streams is important for understanding Tor.

2.3 Connection Input

Tor uses libevent [36] to handle input and output to and from kernel TCP buffers. Tor registers sockets that it wants to read with libevent and configures a notification callback function. When data arrives at the kernel TCP input buffer (Figure 2.1(a)), libevent learns about the active socket through its polling interface and asynchronously executes the corresponding callback (Figure 2.1(b)). Upon execution, the read callback determines read eligibility using token buckets.

Token buckets are used to rate-limit connections. Tor fills the buckets as defined by configured bandwidth limits in one-second intervals while tokens are removed from the buckets as data is read, although changing that interval to improve performance is currently being explored [37]. There is a global read bucket that limits bandwidth for reading from all connections as well as a separate bucket for throttling on a per-connection basis (Figure 2.1(c)). A connection may ignore a read event if either the global bucket or its connection bucket is empty. In practice, the per-connection token buckets are only utilized for edge (non-relay) connections. Per-connection throttling

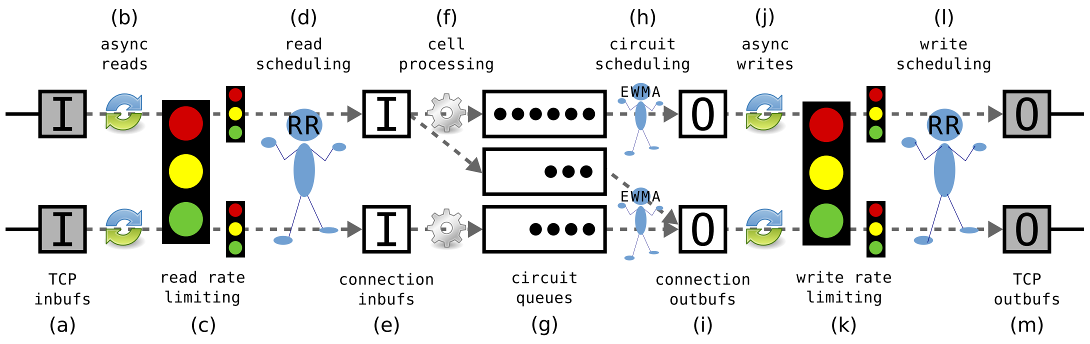


Figure 2.1: A Tor relay's internal architecture.

reduces network congestion by penalizing noisy connections, such as bulk transfers, and generally leads to better performance [38] for most users.

When a TCP input buffer is eligible for reading, a round-robin (RR) scheduling mechanism is used to read the smaller of 16 KiB and $\frac{1}{8}$ of the global token bucket size per connection (Figure 2.1(d)). This limit is imposed in an attempt at fairness so that a single connection can not consume all the global tokens on a single read. However, recent research shows that input/output scheduling leads to unfair resource allocations [39]. The data read from the TCP buffer is placed in a per-connection application input buffer for further processing (Figure 2.1(e)).

2.4 Flow Control

Tor uses an end-to-end flow control algorithm to assist in keeping a steady flow of cells through a circuit. Clients and exit relays constitute the *edges* of a circuit: each are both an ingress and egress point for data traversing the Tor network. Edges track data flow for both circuits and streams using cell counters called *windows*. An ingress edge decrements the corresponding stream and circuit windows when sending cells, stops reading from a stream when its stream window reaches zero, and stops reading from all streams multiplexed over a circuit when the circuit window reaches zero. Windows are incremented and reading resumes upon receipt of `SENDME` acknowledgment cells from egress edges.

By default, circuit windows are initialized to 1000 cells (500 KiB) and stream windows to 500 cells (250 KiB). Circuit `SENDME`s are sent to the ingress edge after the egress edge receives 100 cells (50 KiB), allowing the ingress edge to read, package, and forward 100 additional cells. Stream `SENDME`s are sent after receiving 50 cells (25 KiB) and allow an additional 50 cells. Window sizes can have a significant effect on performance and recent work suggests an algorithm for dynamically computing them [40].

2.5 Cell Processing and Queuing

Data is immediately processed as it arrives in connection input buffers (Figure 2.1(f)) and each cell is either encrypted or decrypted depending on its direction through the circuit. The cell is then switched onto the circuit corresponding to the next hop and placed into the circuit's first-in-first-out (FIFO) queue (Figure 2.1(g)). Cells wait in circuit queues until the circuit scheduler selects them for writing.

2.6 Scheduling

When there is space available in a connection's output buffer, a relay decides which of several multiplexed circuits to choose for writing. Although historically this was done using round-robin, a new exponentially-weighted moving average (EWMA) scheduler was recently introduced into Tor [33] and is currently used by default (Figure 2.1(h)). EWMA records the number of packets it schedules for each circuit, exponentially decaying packet counts over time. The scheduler writes one cell at a time chosen from the circuit with the lowest packet count and then updates the count. The decay means packets sent more recently have a higher influence on the count while bursty traffic does not significantly affect scheduling priorities.

2.7 Connection Output

A cell that has been chosen and written to a connection output buffer (Figure 2.1(i)) causes an activation of the write event registered with libevent for that connection. Once libevent determines the TCP socket can be written, the write callback is asynchronously executed (Figure 2.1(j)). Similar to connection input, the relay checks both the global write bucket and per-connection write bucket for tokens. If the buckets are not empty, the connection is eligible for writing (Figure 2.1(k)) and again is allowed to write the smaller of 16 KiB and $\frac{1}{8}$ of the global token bucket size per connection (Figure 2.1(l)). Data is written to a kernel-level TCP buffer (Figure 2.1(m)) and sent to the next hop.

Chapter 3

Shadow: Running Tor in a Box for Accurate and Efficient Experimentation

3.1 Introduction

Tor's goal to provide *low-latency* anonymity for its clients has led to an enormous amount of research on topics including, but not limited to: anonymity attacks and defenses [41, 17, 19, 20, 42]; system design, performance, and scalability improvements [40, 35, 43, 33, 44]; and the economics of volunteering relays to the Tor network [45, 46, 47]. Most Tor research – whether implementing a new design approach or analyzing a potential attack – either requires or would benefit from access to the live Tor network and the data it generates. However, such access might invade clients' privacy or be infeasible to provide – testing a small design change in the real network requires propagating that change either to hundreds of thousands of Tor clients or to thousands of volunteer relays, and in some cases both. Therefore researchers often use alternative strategies to experiment and test new research designs and proposals.

3.1.1 Tor Experimentation

One approach for experimenting with Tor outside of the live public network is to configure a parallel private test network deployment [41, 33] either using machines at a university or a platform such as PlanetLab [48]. Since live deployments run real software over real hardware, the results are generally accepted. However, PlanetLab and other private deployments do not accurately reflect the same network conditions of the public Tor network, are difficult to manage, and do not scale well – PlanetLab has only around one thousand nodes of which roughly half are usable at any time. Therefore researchers often experiment through simulation [46, 49, 47, 50]. Simulating particular Tor mechanisms may increase scalability, but also harms accuracy: the Tor software and protocols are continuously updated by several Tor developers, causing simulators to become outdated and unmaintained. Moreover, since simulators tend not to be reused, the results of one group may be inconsistent with or can not be verified by other groups.

3.1.2 Tor in a Box

To increase consistency, accuracy, and scalability of Tor experiments, we design and develop a new and unique simulation architecture called Shadow. Shadow allows us to run a private Tor network on a single machine while controlling all aspects of an experiment. Results are repeatable and easily verifiable through independent analysis. Although Shadow *simulates* the network layer, it *links to and runs real Tor software*, allowing us to experiment with new designs by implementing them directly into the Tor source code. This strategy expedites the process of incorporating proposals into Tor since software patches can be submitted to the developers. Shadow is capable of simulating a *large* and *diverse* private Tor network, requiring little to no modification to the numerous supported Tor software versions. Shadow's focus on usability and commitment to open source software[1] improves accessibility and promotes community adoption, interactions, and contributions.

Shadow is a discrete-event simulator that utilizes techniques allowing it to run real applications in a simulation environment. Real applications are encapsulated in a plug-in wrapper that contains functions necessary to allow Shadow to interact with the application. Although the application is only loaded into memory once, the plug-in registers memory addresses for all variable application state and Shadow manages a copy of these memory regions for each node in the simulation. Similar to a kernel context switch, Shadow swaps in the current node's version of this state before passing control to the application, and swaps out the state when control returns. Function interposition allows Shadow to intercept function calls, e.g. socket and event library calls, and redirect them to a simulated counterpart. As detailed in Section 3.3, we run Tor using these techniques, as well as symbol table manipulations, without modifying the source code.

3.1.3 Accurate Simulation

We validate Shadow's accuracy against a 402-node PlanetLab deployment, testing network performance using HTTP file transfers both directly and through a private Tor network deployment. Although private Tor networks on PlanetLab do not consistently represent the live Tor network well, they allow us to test our ability to model a real, diverse network (i.e. how well we can "shadow" PlanetLab conditions). We find that our results are within reason although PlanetLab exhibits highly variable behaviors because of overloaded CPUs caused by co-location and resource sharing.

To validate Shadow's ability to accurately and consistently represent the real, live

[1]Shadow source code is publicly available under the GPLv3 [51, 52].

Tor network, we simulate a 1051-node topology with bandwidth and relay characteristics taken from a live Tor network consensus. We model the Internet using network latency measurements taken between all PlanetLab nodes. We find that client performance in Shadow closely matches live statistics gathered by the Tor Project [30], with download time quartiles within 15 percent of the live statistics for various download sizes.

3.2 Requirements

3.2.1 Accuracy

In order to produce results that are consistent with and representative of the live Tor network, Shadow should run a minimally-modified version of the native Tor software. Running the Tor software in our simulator will ensure that Tor's behavior in our simulated Tor network will closely represent the behavior of Tor in the live network.

In addition to running the Tor software, Shadow should also have accurate models of system-level interactions. Tor is mostly concerned with buffering, encrypting/decrypting cells, and sending and receiving large amounts of network traffic with non-blocking I/O. Inaccurate models of these mechanisms would lead to inaccurate results and measurements of Tor's behavior. Therefore, we are model the system-level network stack of an operating system by simulating TCP and UDP, correctly managing network-level buffers and buffer sizes, and simulating non-blocking event-driven I/O. Since a large amount of Tor's run-time is spent performing cryptography and processing data, Shadow should avoid execution of expensive cryptographic operations while instead modeling the CPU delays that would have occurred had the cryptography actually been performed.

Finally, accurate software and an accurate system will not function correctly without an accurate network. First, Shadow requires models for network characteristics including latency and reliability of network links, complex AS-level topologies, and upstream/downstream capacities for end-hosts. Second, Shadow must accurately model the network characteristics of Tor, including relay-contributed bandwidth, faithful bandwidth distributions among entry, middle, and exit relays, and geographical distribution of relays. Shadow must also incorporate network traffic from Tor clients and model accurate distributions of that traffic from live Tor traffic patterns.

3.2.2 Usability and Accessibility

A simulator that produces accurate results characteristic of the live Tor network will be of little use to the community without a usable simulation framework. Shadow should therefore do the following to increase usability and promote community adoption.

First, Shadow should be simple to obtain, build, and configure to allow for rapid deployment. Users should be able to run a simulation with minimal overhead and little or no configuration. However, advanced users should be able to easily modify a simulation, generate new topologies, and configure network and system parameters. Simulation results should be easy to gather and parse to produce visualizations that allow an analysis of the network state. Second, Shadow should run completely as a user-level process on a single machine with inexpensive hardware to minimize overhead costs associated with obtaining, configuring, and managing multiple machines or clusters. Shadow should be publicly accessible so that results can be easily compared and verified.

3.3 Design

Shadow is a discrete event simulator that can run real applications as plug-ins while requiring minimal modifications to the application. Plug-ins containing applications link to Shadow libraries and Shadow dynamically loads and natively executes the application code while simulating the network communication layer. Shadow was originally[2] a fork of Distributed Virtual Network (DVN) simulator [53], adding roughly 18,000 lines of code (including example plug-ins). An overview of Shadow's design is depicted in Figure 3.1 and details about its core simulation engine are given below in Section 3.3.1.

Shadow dynamically loads plug-ins and instantiates virtual nodes as specified in a simulation script. Communication between Shadow and the plug-in is done through a well-defined callback interface implemented by the plug-in. When the appropriate callback is executed, the plug-in may instantiate and run its non-blocking application(s). The application will cause events to be spooled to the scheduler by executing a system call that is intercepted by Shadow and redirected to a function in the node library. The interceptions allow integration of the application into the simulation environment

[2]The description of Shadow in this chapter is based on Shadow v1.0.0. All code inherited from DVN has been removed in newer versions of Shadow.

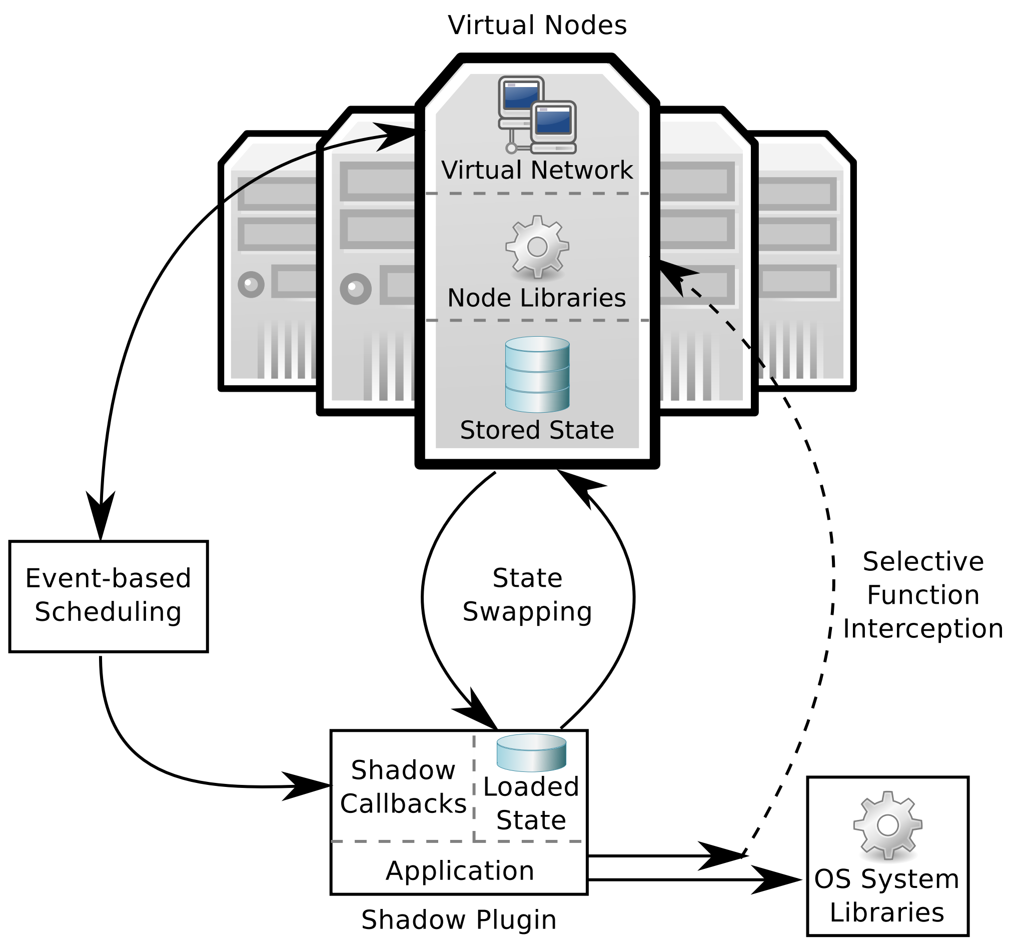


Figure 3.1: Shadow's architectural design. Using a plug-in wrapper, real-world applications are integrated into Shadow as virtual nodes while system and library calls are intercepted and replaced with Shadow-specific implementations.

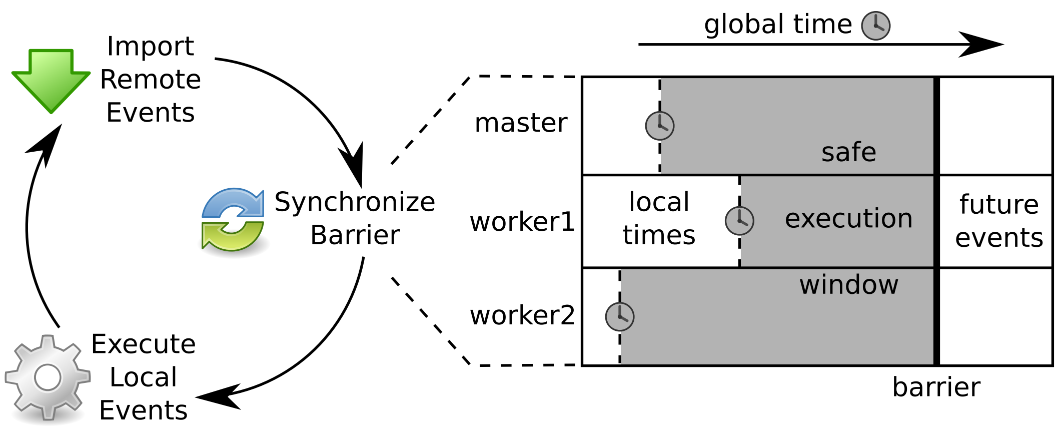


Figure 3.2: Main loop and conservative multi-process synchronization using dynamic barriers. Safe execution windows are calculated using the minimum local worker time plus the minimum simulated latency between nodes. The barrier is dynamically pushed as local times advance.

without requiring modification of application code. Virtual nodes communicate with each other through a virtual network which spools packet and other network related events to the scheduler. Each virtual node stores only application-specific state and loads/unloads the state as necessary during simulation execution. We now describe the main architectural components that enable Shadow to realize the above functionality and fulfill our design requirements discussed in the previous section.

3.3.1 Core Simulation Engine

Shadow was originally a fork of the Distributed Virtual Network (DVN) Simulator [53]. DVN is a discrete event, multi-process, scalable UDP-based network simulator written in C that can simulate hundreds of thousands of nodes in a single experiment. DVN takes a unique approach to simulation by running UDP-based user applications as modules loaded at run-time. Among DVN's core components are the per-process event schedulers, a process synchronization algorithm, and a module subsystem. We describe each of these main components but note that Foo Kune *et al.* [53] provide DVN's design details in much greater resolution.

Discrete-event Scheduler

DVN implements a conservative, distributed scheduling algorithm (see Figure 3.2) that utilizes message queues to transfer events between workers. The scheduling algorithm consists of three phases: importing events initiated from remote nodes, synchronizing worker processes, and executing local node events. During the import phase, workers process incoming messages containing events and store them in a custom local event priority queue. After all messages are imported, workers send synchronization messages (discussed below) to other workers and finally process local events in non-decreasing order. Incoming messages are buffered while processing local events and handled by the scheduler during the next import phase.

Multi-process Synchronization

Messages between the master and workers enable global time synchronization throughout the simulation. Synchronized time is vital to ensure events are executed in the correct order since a conservative scheduling algorithm cannot revert events. By exchanging messages, each process tracks the local time of all other processes. A *barrier* is computed by taking the minimum local time of each process and adding the minimum network latency between any two network nodes in the simulation. The barrier represents the earliest possible time that an event from one process may affect another process. Each process may execute events in its local event queue as long as the event execution time is earlier than the barrier. This is called the *safe execution window*: any event in this window may be safely executed without compromising the order of events (i.e. time will never jump backwards to execute a past event). Barriers are dynamically updated as new synchronization messages update local times. Future events are allowed to execute as the barrier progresses through time. This synchronization approach allows the distribution of events to multiple processes.

Module Subsystem

DVN contains a subsystem for dynamically loading modules. Modules, pieces of code that are run by nodes, are generally created by porting application code to use DVN network calls and implementing special functions required by DVN. The special functions allow modules to receive event callback notifications from DVN. Although each module may be run by several nodes, module libraries are only loaded into memory once. In order to support multiple nodes running the same module, DVN requires each module to register all variable application state. Using the registered memory addresses, DVN may properly load node state before passing execution control to the module, and store node state after regaining control.

3.3.2 Simulation Script

Each simulation is bootstrapped with a simulation script written in a custom scripting language. This script gives the user access to commands that allow Shadow to dynamically load multiple plug-ins, create and connect networks, and create nodes. Valid plug-ins are loaded by supplying a filepath while parameters such as latency, upstream and downstream bandwidth, and CPU speed are specified by either loading a properly formatted CDF data file or generating a CDF using a built-in CDF generator. Hostnames may be specified for each node and are otherwise automatically generated to facilitate support for a Shadow name service. The script also specifies which plug-in to run and when to start each node.

Events are extracted from a properly formatted simulation script and spooled to the event scheduler using the times specified with each command. After the script is parsed, the simulation begins by executing the first extracted event and runs until either there are no events remaining in the scheduler or the end time specified in the script is reached. Each node creation event triggers the allocation of a virtual node and its network and culminates in a callback to a Shadow plug-in for application instantiation.

3.3.3 Shadow Plug-ins

A Shadow plug-in is an independent library that contains applications the user wishes to simulate and a wrapper around these applications allowing integration with the Shadow simulation environment. Each Shadow plug-in wrapper implements the plug-in interface – a set of callbacks that Shadow uses to communicate with the plug-in. Plug-ins may also link to a special Shadow plug-in utility libraries to, e.g., resolve a hostname or IP address or log messages.

Application

To run in Shadow, an application must be asynchronous, i.e. non-blocking, to prevent simulator deadlocks during the execution of application code. We note that asynchronicity may be achieved with a small amount of code in the plug-in wrapper that utilizes the built-in Shadow callbacks or by utilizing the `libevent-2.0` asynchronous event library [36], as Shadow supports a subset of this library.

Next, the application must be run as a single process and in a single thread. Child processes or threads forked or spawned by an application will not be properly contained in the simulation environment and are therefore currently unsupported. In most cases forking or spawning children will lead to undefined behavior or undesirable results. We note that most multi-threaded applications have a single-threaded mode and the difficulty in porting those that do not is application-specific.

Finally, the plug-in must register all variable application state with Shadow to facilitate multiple virtual nodes running the same application. Plug-ins fulfill this requirement by passing pointers to node-specific allocated memory chunks and their sizes to a Shadow library function. Therefore each variable must be globally visible during the registration process. However, we note that a plug-in may use standard tools to scan and globalize symbols present in the binary after the linking process. As in our Tor plug-in (Section 3.4), this technique may be used to dynamically generate registration code and eliminates the requirement of modifying variable definitions inside the application.

Shadow Callbacks

The Shadow plug-in interface allows Shadow not only to notify the plug-in when it should allocate and deallocate resources for running the application(s) contained in the plug-in, but also to notify the plug-in when it may perform network I/O (reading and writing) on a file descriptor without blocking. The I/O callbacks are crucial for asynchronicity as they trigger application code execution and prevent applications from the need for polling a file descriptor. The Shadow plug-in library also offers support for a generic timer callback so plug-ins may create additional events throughout a simulation. Note that callbacks may also originate from the virtual event library, as described in Section 3.3.4 below, if the application uses `libevent-2.0`.

3.3.4 Virtual Nodes

In Shadow, a virtual node represents a single simulated host. A virtual node contains all state that is specific to a host, such as addressing and network information that allows it to communicate with other hosts in the network. Virtual nodes also contain Shadow-specific implementations of system libraries that promote homogeneity between existing interfaces. Function interposition allows for seamless integration of applications into Shadow by redirecting calls to system functions to our Shadow implementation. Virtual nodes store their own application-specific state and swap this state into the plug-in's address space before passing control of code execution to the plug-in.

Virtual Network

In Shadow, the virtual network is the main interface through which virtual nodes may communicate. Upon creation, each node's virtual network interface is assigned an IP address and receives upstream and downstream bandwidth rates as configured in the simulation script. Each virtual network contains a transport agent that implements a leaky bucket (i.e. token bucket) algorithm that allows small traffic bursts but ensures average data rates conform to the configured rate. The transport agent handles both incoming and outgoing packets, allowing for asymmetric bandwidth specifications. The agent provides traffic policing by dropping (and causing retransmission of) all non-conforming packets. Conforming incoming packets are passed to the virtual socket library (discussed below) for processing, while events are created for conforming outgoing packets and spooled to the scheduler for delivery to another node after incorporating network latency.

Node Libraries

Each virtual node implements several system functions as well as network, event, and cryptography libraries. *Function interposition* is used to redirect standard system and library functions calls made from the application to their Shadow-specific counterparts. Function interposition is achieved by creating a *preloaded* library with functions of the same name as the target functions, and setting the environment variable `LD_PRELOAD` to the path of the preload library. Every time a function is called, the preload library is first checked. If it contains the function, the preloaded function is called – otherwise the standard lookup mechanisms are used to find the function. No additional modifications are required to hook into Shadow.

Virtual System. The virtual system library implements standard system calls whose results must be modified due to the simulation environment. Functions for obtaining system time are implemented to return the simulation time rather than the wall time and functions for obtaining hostname and address information are intercepted to return the hostnames as defined in the simulation script configured by the user.

The virtual system also contains a virtual CPU module in an attempt to consider processing delays produced by an application. Using a virtual CPU and processing delays improves Shadow's accuracy since without it, all data is processed by the application at a single discrete instant in the simulation. When a virtual node reads or writes data between the application and Shadow, the virtual CPU produces a delay for processing that data. This delay is "absorbed" by the system by delaying the execution of every event that has already been scheduled for that virtual node. As virtual nodes read and write more data, the wait time until the next event increases.

We determine appropriate CPU processing speeds as follows. First, throughput is configured for each virtual CPU – the number of bytes the CPU can process per second. Modeling the speed of a target CPU is done by running an `OpenSSL` [54] speed test on a real CPU of that type. This provides us with the raw CPU processing rate, but we are also interested in the processing speed of an application. By running application benchmarks on the same CPU as the `OpenSSL` speed test, we can derive a ratio of CPU speed to application processing speed. The virtual CPU converts these speeds to a time per-byte-processed and delays its events appropriately.

Virtual Sockets. The Shadow virtual socket library, the heart of the node libraries, implements the most significant features for a Shadow simulation. The virtual socket library implements all system socket functionality which includes: creating, opening, and closing sockets; sending, buffering, and receiving data; network protocols like the *User Datagram Protocol* (UDP) [55] and the *Transmission Control Protocol* (TCP) [56]; and other socket-level functionalities. Shadow's tight integration of socket functionalities and strong adherence to the RFC specifications results in an extremely accurate network layer as we'll show in Section 3.5.

Shadow intercepts and redirects functions from the system socket interface to the Shadow-specific virtual socket library implementation. When the application sends data to the virtual socket library, the data is packaged into packet objects. The packaging process copies the user data only twice throughout the lifetime of the packet, meaning the same packet object is shared among nodes. Only pointers to the packet are copied as the packet travels through various socket and network buffers, although buffer sizes are computed using the full packet size.

Our virtual socket libraries implement socket-level buffering, data retransmission, congestion and flow control mechanisms, acknowledgments, and TCP auto-tuning. TCP auto-tuning is required to correctly match buffer sizes to connection speeds since neither high bandwidth connections with small network buffers nor low-bandwidth connections with large network buffers will achieve the expected performance. TCP auto-tuning allows network buffers to be dynamically computed on a per-connection basis, allowing for highly accurate transfer rates even when endpoints have asymmetric bandwidth.

Virtual Events. Shadow supports the use of `libevent-2.0` [36] to facilitate asynchronous applications while easing application integration. While applications are not required to use `libevent-2.0`, doing so will likely reduce the complexity of the integration process. Shadow intercepts and redirects functions from the `libevent-2.0` interface to the Shadow-specific virtual event library implementation. The virtual event library consists of two main components: an event manager and a virtual I/O monitor. The event manager creates and tracks events and executes event callbacks while the I/O monitor tracks the state of Shadow buffers, informing the manager when a state change may require an event callback to fire for a given file descriptor.

Virtual Cryptography. Simulating an application that performs cryptography offers a chance for reducing simulation run-time. As data is passed from virtual node to virtual node during the simulation, in most cases it is not important that the data is encrypted: since we are not sending data out across a real network, confidentiality is not necessarily required. Therefore, applications need not perform expensive encryption and decryption operations during the simulation, saving CPU cycles on our simulation host machine.

Shadow removes cryptographic processing by preloading the main `OpenSSL` [54] functions used for data encryption. The `AES_encrypt` and `AES_decrypt` functions are used

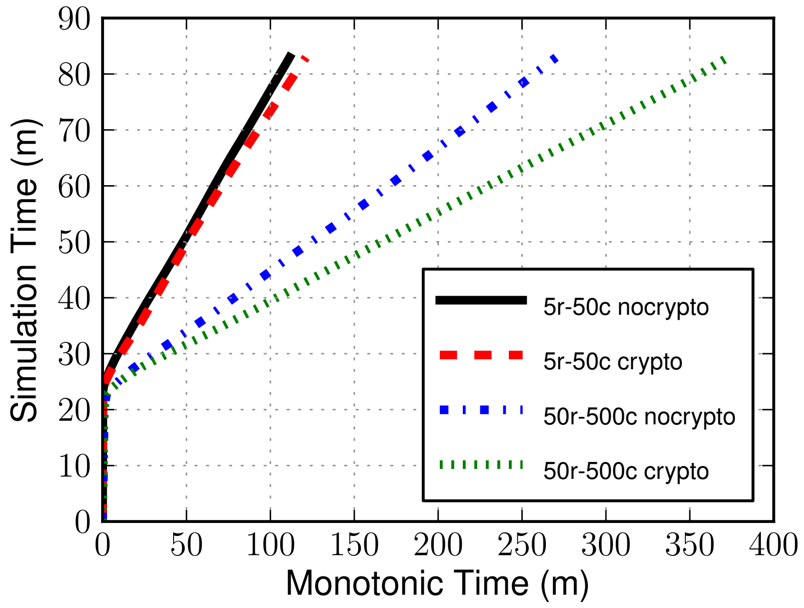


Figure 3.3: Simulation vs. wall clock time. Skipping expensive cryptographic operations results in a linear decrease in experiment run-time – nearly a one-third reduction in run-time for a small, 550-node Tor experiment.

for bulk data encryption and the `EVP_Cipher` function is used to secure data on SSL/TLS connections. These functions only perform the low-level cipher operations: all other supporting cryptographic functionality is unmodified. When preloading these functions, Shadow will not perform the cipher operation during encryption and decryption. Our virtual CPU already models application processing delays, and skipping the cipher operations will not affect application functionality.

Figure 3.3 shows the time savings Shadow realizes using this technique with the Scallion plug-in (discussed below in Section 3.4) for various Tor network sizes. Larger savings in real running time are realized as experiment size increases.

Stored State

Multiple virtual nodes may run the same plug-in. Rather than duplicating the entire plug-in in memory for each virtual node, Shadow only duplicates the variable state – the state of an application that will change during execution. Registration of this variable state with Shadow happens once for each plug-in. The plug-in registration procedure allows Shadow to determine which memory regions (beginning address and length) in the current address space will be modified by each virtual node running the plug-in.

Following registration, Shadow possesses pointers to each memory region that may be changed by the plug-in or application. Multiple nodes for each plug-in are supported by allocating node-specific storage for each registered memory region and maintaining a copy of each plug-in's state. For transparency, Shadow loads a node's state before every context switch from Shadow to the plug-in, and saves state back to storage when the context switches back to Shadow. This process minimizes the total memory consumption of each plug-in, and results in significant memory savings for large simulations and applications.

3.4 The Scallion Plug-in: Running Tor in Shadow

Shadow was designed especially for running simulations using the Tor application. Therefore, Shadow design choices were made in support of "Scallion"[3], a Tor plug-in implementation. Each virtual node running the Scallion plug-in represents a small piece of the Tor network. Since Shadow supports most functionality needed by Scallion, the plug-in implementation itself is minimal (roughly 1500 lines of code). Here we describe some of the specific components necessary for the Tor application plug-in.

3.4.1 State Registration

Recall that Shadow requires all variable application state to be registered for replication among virtual nodes. Scallion must find and register all Tor variables, including static and global variables. Unfortunately, static variables are not accessible outside the scope in which they were defined. Therefore, scallion uses standard binary utilities such as `objcopy`, `readelf`, and `nm` to dynamically scan, rename, and globalize Tor symbols. Registration code is then dynamically generated based on the symbols present in the Tor object files, and injected into the plug-in before compilation. Note that the size of each variable is also extracted with the binary utilities.

[3]Scallions are onion-like plants with underdeveloped bulbs.

3.4.2 Bandwidth Measurements

TorFlow [57] is a set of scripts that run in the live Tor network, continuously measuring bandwidth of volunteer relays by downloading several files through each. TorFlow helps determine the bandwidth to advertise in the public consensus document. Scallion contains a component that approximates this functionality. However, Scallion need not perform actual measurements since the bandwidth of each virtual node is already configured in Shadow. Scallion queries for these bandwidth values through a Shadow plug-in library function and writes the appropriate file that is used by the directory authorities while computing a new consensus. The `V3Bandwidth` file is updated as new relays join the simulated Tor network.

3.4.3 Tor Preloaded Functions

In an effort to minimize the amount of changes to Tor, Scallion utilizes the same function interposition technique as Shadow. Scallion may intercept any Tor function for which it requires changes and implement a custom version. Changes in Tor are required only if the target function is static, in which case Tor can be modified to remove the static specifier. We now discuss some functional differences between Tor and Scallion.

The Tor socket function wrapper is one function that is intercepted by Scallion and modified to pass the `SOCK_NONBLOCK` flag to the socket call since Shadow requires non-blocking sockets. Another modification involves the Tor main function, which is not suitable for use in Scallion since it contains an infinite loop. This function is extracted to prevent the simulation from blocking, and Scallion instead relies on event callbacks from Shadow to implement Tor's main loop functionality.

Tor is a multi-threaded application, launching at least one CPU worker thread to handle onionskin tasks – peeling off or adding a layer of encryption – as they arrive from the network. Scallion implements an event-driven version of Tor's CPU worker since Shadow requires a single-threaded, single-process application. This is done by intercepting the Tor function that spawns a CPU worker and relying on the virtual event library to execute callbacks when the CPU worker has data ready for processing. The CPU worker performs its task as instructed by Tor, and communicates with Tor using a socket pair (a virtual pipe) as before. The virtual event library simplifies the implementation of the CPU worker and the functionality it provides.

Finally, Scallion intercepts Tor's bandwidth reporting function. Each Tor relay reports its recent bandwidth history to the directory authorities to help balance bandwidth across all available relays. However, relays' reports are based on the amount of data it has recently transferred, and the reported value is updated every twenty minutes only if it has not changed significantly from the last reported value. This causes relays to be underutilized when first joining the network, and causes bootstrapping problems in new networks since every node's bandwidth will be zero for the first twenty minutes of the simulation. Without appropriate bandwidth values, clients no longer perform weighted relay selection and instead choose relays at random. To mitigate these problems, Scallion intercepts the bandwidth reporting function and returns its configured BandwidthRate no matter how much data it has transferred. This improves bootstrapping and path selection for the simulated Tor network.

3.4.4 Configuration and Usability

There are several challenges in running accurate Tor network simulations with the Scallion plug-in and Shadow. Although Shadow minimizes the memory requirements, running several instances of Tor still requires an extremely large amount of memory. Therefore, simulations must generally run with scaled-down versions of Tor network topologies and client-imposed network load.

Correctly scaling available relay bandwidth and network load is complicated. For example, several relays with smaller bandwidth capacities will not result in the same network throughput as fewer relays with larger bandwidth capacities, even if the total capacities are equal. Further, correctly distributing this bandwidth among entry, middle, and exit nodes can be tricky. Although live Tor consensus documents may be used to assist in network scaling, two randomly generated consensus topologies can have drastically different network throughput measurements. Network throughput also depends on the number of configured clients and the load they induce on Tor. Published results about client-to-relay ratios [27] and protocol-level statistics [28] can only be used as a rough guide to creating clients and inducing the correct load. When generating a scaled topology, it is essential that performance measurements of simulations be compared to live Tor statistics for accuracy.

Due to these challenges, we implemented a script to generate and run simulations given a network consensus document. The script parses the consensus document and randomly selects relays based on configurable network sizes. Configurable parameters include the fraction of exit relays to normal relays, number of clients, and client type distributions. The script eases the generation of accurate scaled topologies and drastically improves simulator usability.

3.5 Verifying Simulation Accuracy

Many aspects of Shadow's design (discussed in Section 3.3) were chosen in order to produce accurate simulations. We now perform several experiments to verify Shadow's accuracy.

3.5.1 File Client and Server Plug-ins

`HTTP` client and server plug-ins were written for Shadow in order to provide a mechanism for transferring data through the Shadow virtual network. These plug-ins also include support for a minimal `SOCKS` client and proxy. The client may download any number of specified files with configurable wait times between downloads while the server supports buffering and multiple simultaneous connections. These plug-ins are used to test network performance during a simulation. Stand-alone executables using the same code as the plug-ins are also compiled so that client and server functionality on a live system and network is identical to Shadow plug-in functionality.

3.5.2 PlanetLab Private Tor Network

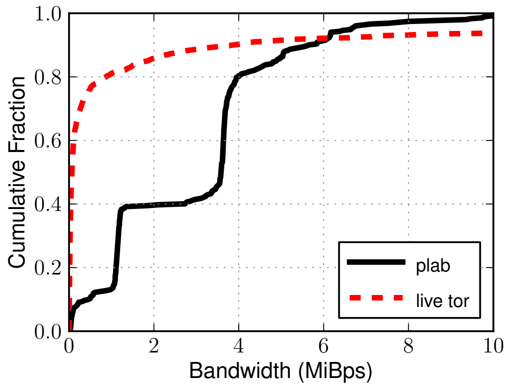
In order to verify Shadow's accuracy, we perform experiments on PlanetLab. Our experiments consist of file clients and servers running the software described above in Section 3.5.1. In our first PlanetLab experiment, each of 361 `HTTP` clients download files directly from one of 20 `HTTP` servers, choosing a new server at random for each download. 18 of the 361 clients approximate a bulk downloader, requesting a 5 MiB file immediately after finishing a download while the remaining 343 clients approximate a web downloader, pausing for a short time between 320 KiB file downloads. The length of the pause is drawn from the UNC think-time distribution [58] which represents the time between clicks for a user browsing the web (the median pause is 11 seconds). Clients track both the time to receive the first byte of the data payload and the time to receive the entire download. We selected the fastest PlanetLab nodes (according to the bandwidth tests described below) as our `HTTP` servers to minimize potential server bottlenecks, although we note that fine-grained control is complicated by PlanetLab's dynamic resource adjustment algorithms.

Our second PlanetLab experiment is run exactly like the first, except all downloads are performed through a private PlanetLab Tor network consisting of 16 exit relays, 24 non-exit relays, and one directory authority. All `HTTP` clients also run a Tor client and proxy their downloads through Tor using a local connection to the Tor `SOCKS` server.

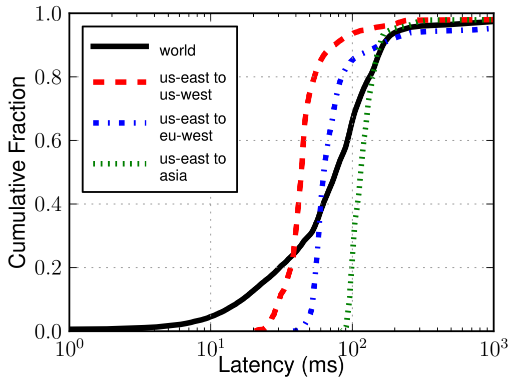
Shadowing PlanetLab

To replicate the PlanetLab experiments discussed in Section 3.5 in Shadow, we require measurements of PlanetLab node bandwidth, latency between nodes, and an estimate of node CPU speed. These measurements allow us to configure virtual nodes and a virtual network that approximates PlanetLab and network conditions typical of the Internet. First, we estimate PlanetLab node bandwidth by performing Iperf [59] bandwidth tests from each node to every other node. We estimate a node's bandwidth as the maximum upload rate to any other node.

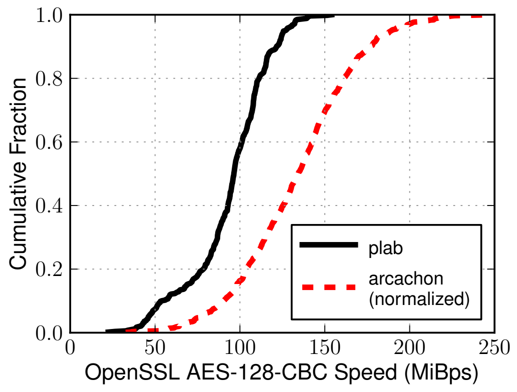
Figure 3.4(a) shows the results of our measurements compared with available bandwidth from Tor relays extracted from the Tor network status consensus. Notice the sharp increase in the number of nodes with 1.25 MiB/s (10 Mb/s) and 3.75 MiB/s (30 Mb/s) connections. PlanetLab rate-limiting is the likely reason: the most popular node-defined limit is 10 Mb/s while PlanetLab also implements a fair-sharing algorithm by distributing opportunistic fractions of bandwidth to active slices. Also notice that our PlanetLab distribution does not approximate the live Tor distribution well, which means that our measurements in this experiment will not provide a good indication of the performance of the live Tor network. Recall, however, that our focus here is *accurately shadowing PlanetLab*: re-creating a network consistent with live Tor is explored below in Section 3.5.3.



(a) PlanetLab vs. Tor Bandwidth



(b) PlanetLab Regional Latency



(c) CPU Throughput

Figure 3.4: (a) Bandwidth measurements of PlanetLab nodes and live Tor relays. Relay bandwidth values were taken from a live consensus. (b) Latency between PlanetLab nodes, shown as aggregate ("world") and inter-region latency measurements. (c) Measured CPU speeds for PlanetLab nodes and our Intel Core2 Duo lab machine *arcachon*. The results from arcachon were normalized to create a distribution usable in Shadow.

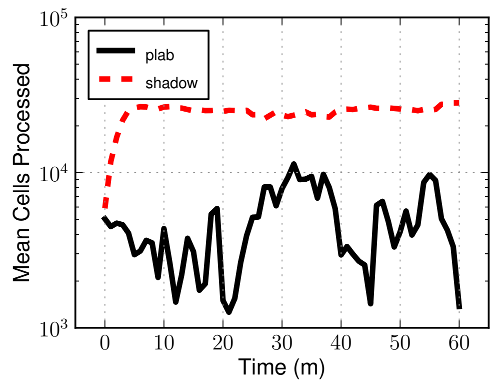
To model network delays due to propagation and congestion, we perform latency estimates between all pairs of nodes using the Unix command `ping`. The aggregate results of world latencies are shown in Figure 3.4(b). Deriving a network model and topology from the latency measurements is a bit more complex since it depends on the geographical location of the source and destination of a ping. We approximate a network model by creating nine geographical regions and placing each node in a region using a GeoIP lookup [60]. We then create a total of 81 CDFs representing all possible inter- and intra-region latencies. We configure nine virtual networks in Shadow and connect them into a complete graph topology, where latencies for packets traveling over each link are drawn from the corresponding CDF. Latencies for a few selected regions are also shown in Figure 3.4(b).

Finally, we measure CPU speed of each node in order to accurately configure delays for Shadow's virtual CPU system described in Section 3.3.4. As in our previous description, `OpenSSL` speed tests are run to get raw CPU throughput for PlanetLab nodes. Since PlanetLab nodes are often constrained, we also created a normalized distribution based on the CPU speed of *arcachon* – a standard desktop machine in our lab. CPU throughput is shown in Figure 3.4(c). Tor application throughput – measured by benchmarks in which the middle relay is configured with a bandwidth bottleneck – is combined with raw CPU throughput measurements to configure each node's virtual CPU delay.

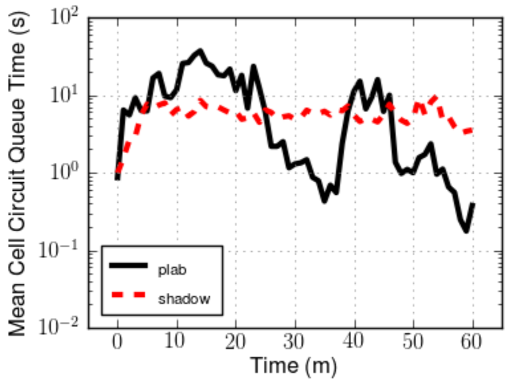
Client Performance

Figure 3.5 shows the results of our PlanetLab and Shadow experiments. We are mainly interested in two metrics: the *time to receive the first byte* of the data payload (ttfb) and the *time to complete* a download (dt). The ttfb metric provides insight into the delays associated with sending a request through multiple hops and the responsiveness of a circuit, and also represents the minimum time a web user has to wait until anything is displayed in the browser. The dt metric is a measurement of overall client performance.

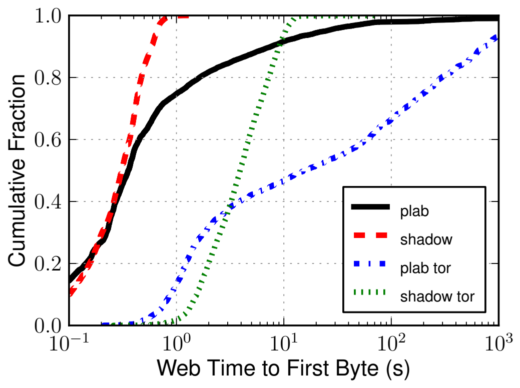
Figures 3.5(c) and 3.5(e) show the ttfb metric for web and bulk clients with direct and Tor-proxied requests both in PlanetLab and Shadow. Downloads through Tor take longer than direct downloads, as expected, since data must be processed and forwarded by multiple relays. Shadow seems to closely approximate the *network conditions* in



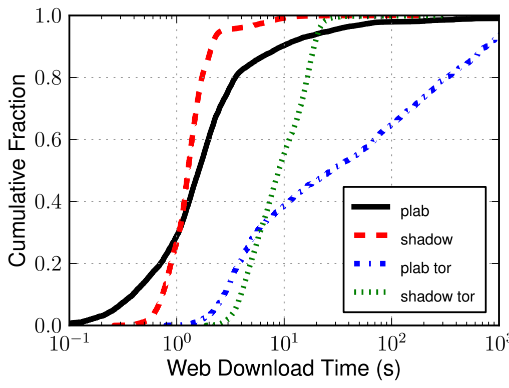
(a) Cell Processing Time



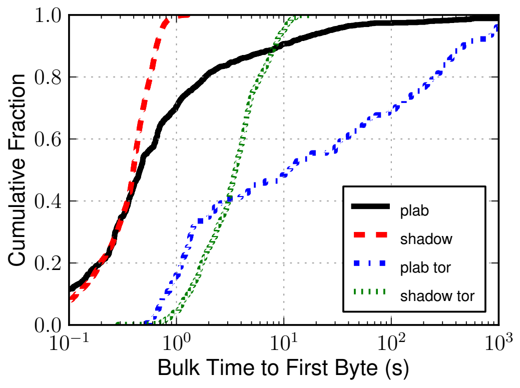
(b) Cell Queuing Time



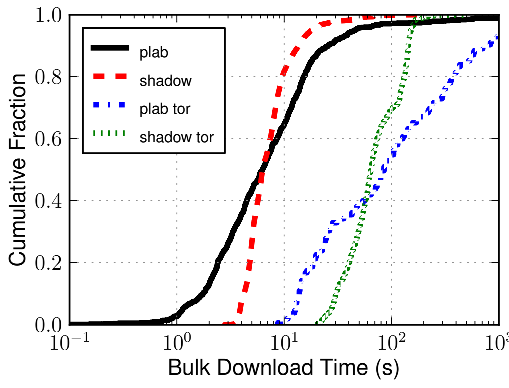
(c) 320 KiB, First Byte



(d) 320 KiB, Last Byte



(e) 5 MiB, First Byte



(f) 5 MiB, Last Byte

Figure 3.5: Shadow and PlanetLab network performance. PlanetLab download experiments were run with and without Tor and mirrored in Shadow. PlanetLab results show higher variability due to co-location and network/hardware interruptions.

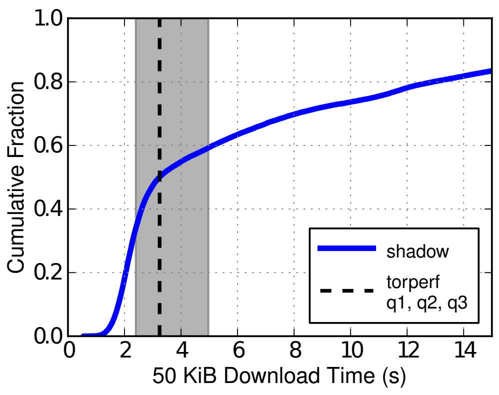
PlanetLab, as shown by the close correspondence between the lower half of each CDF. However, PlanetLab exhibits slightly higher variability in ttfb than Shadow as seen in the tail of the plab and shadow CDFs – a problem that is exacerbated when downloads are proxied through Tor. Higher variability in results is likely caused by increased PlanetLab node delay due to resource contention with other co-located experiments.

Figures 3.5(d) and 3.5(f) show similar conclusions for the dt metric. Shadow results appear off by a small factor while we again see higher variability in download completion times for PlanetLab. However, inaccuracies in download times appear somewhat independent of file size. As shown in Figure 3.5(a), statistics gathered from Tor relays support our conclusions about higher variability in delays. Shown is the number of processed cells for each relay over the one hour experiment and the one-minute moving average. The moving average of processed cells is slightly higher for Shadow because of PlanetLab's resource sharing complexity while the individual relay measurements also show higher variability for PlanetLab. Figure 3.5(b) shows that Shadow queue times are very close to those measured on PlanetLab, and again shows PlanetLab's high variability. While we are optimistic about our conclusions, we emphasize that PlanetLab results should be analyzed with a careful eye due to the issues discussed above.

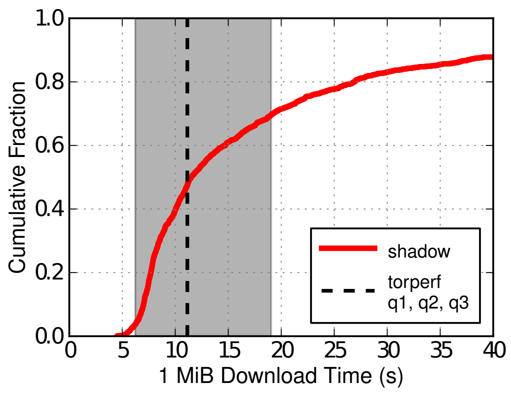
3.5.3 Live Public Tor Network

Although the PlanetLab results show how Shadow performance compares to that achieved while running on PlanetLab and a *private* Tor network, they do not show how accurately Shadow can approximate the live *public* Tor network containing thousands of relays and hundreds of thousands of clients geographically distributed around the world. Therefore, we perform a separate set of experiments to test Shadow's ability to approximate live Tor network conditions as documented by The Tor Project [30]. Comparing results with statistics from Tor Metrics gives us much stronger evidence of Shadow's ability to accurately simulate the live Tor network.

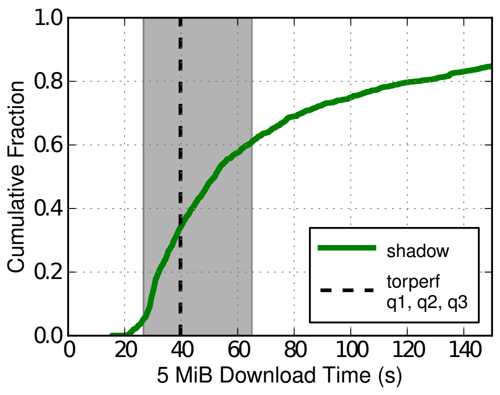
The experiments are similar to those performed on PlanetLab: web and bulk clients download variable-sized files from servers through a private Tor network. However, file sizes are modified to 50 KiB, 1 MiB, and 5 MiB as used by TorPerf while configuration of Shadow nodes is also slightly modified to approximate resources available in live Tor. In these experiments, we use a directory authority, 50 relays, 950 web clients,



(a) 50 KiB File Downloads



(b) 1 MiB File Downloads



(c) 5 MiB File Downloads

Figure 3.6: Shadow-Tor compared with live-Tor network performance. TorPerf represents live Tor network performance statistics available at `metrics.torproject.org`. The gray area shows TorPerf first to third quartile stretch while the dotted line represents the TorPerf median. Shadow closely approximates the expected Tor client performance for all file sizes.

50 bulk clients, and 200 servers. We use a live Tor consensus[4] to obtain bandwidth limits for Tor relays and ensure that we correctly scale available bandwidth and network size, while client bandwidths are estimated with 1 MiB down-link and 3.5 MiB up-link speeds (not over-subscribed). Each relay is configured according to the live consensus: a `CircuitPriorityHalflife` of 30, a 40 KiB `PerConnBWRate`, and a 100 MiB `PerConnBWBurst`. Geographical location and latencies are configured using our PlanetLab dataset [52].

Figure 3.6 shows Shadow's accuracy while simulating a shadow of the live Tor network. CDFs of Shadow download completion times for each file size are compared with download times measured and collected by The Tor Project. The gray area represents the first-to-third quartile stretch and the dotted line shows the median download time extracted from live Tor network statistical data available at The Tor Metrics Portal [30] (gathered during April 2011 – the same month as our consensus). To maximize accuracy, the left edge of the gray area should intersect the CDF at 0.25, the right edge at 0.75, and the dotted line at 0.5. Our results show that the median download times are nearly identical for 50 KiB and 1 MiB downloads and within ten percent for 5 MiB downloads while the first and third quartiles are within 15 percent in all cases. We believe these results provide strong evidence of Shadow's ability to accurately simulate Tor. Further, we've shown that we can correctly scale down the Tor network in our simulations while maintaining the performance properties of the live Tor network.

3.6 Limitations

Shadow is a discrete event simulator. By definition, Shadow imitates the behavior of system and network processes. These imitations were done by exploring and measuring real systems and real networks to produce models of real behaviors suitable for our study of performance in Tor. This modeling approach fundamentally limits the ability to adapt to highly dynamic environments, potentially reducing simulation accuracy. We now briefly discuss our modeling approaches in the context of their effects on simulations to emphasize the importance of analyzing and verifying results so that

[4]The consensus was retrieved on 2011-04-27 and valid between 03:00:00 and 06:00:00.

future researchers may make informed decisions regarding experimentation with their algorithms and protocols.

In addition to simulating TCP, Shadow models the Internet by utilizing real delays gathered from `ping` measurements on PlanetLab. The measurements give us a distribution of pairwise delays between nodes. We then categorize all nodes into geographical "regions" and aggregate node distributions between each region. This approximation forms our model of the Internet and is given as input to a simulation: when sending a packet from a node in one region to a node in another, we sample the distribution corresponding to the link between those regions. This model is based on specific measurements between specific points at a specific time. If Internet congestion at that time was uncharacteristically high or low, our model would skew results. Although this approach seemed to provide an adequate delay model in our simulations, future work should consider if their experiments could benefit from something more robust. In particular, experiments that depend on side-channels stemming from latency, throughput, or the inner workings of a specific TCP implementation may require an alternative experimentation approach.

Scallion models system processing delays by gathering PlanetLab `OpenSSL` speed and application performance test measurements. Processing delays are then accumulated as the application reads and writes data, as this allows Shadow to quantify the amount of work the application is performing. Delays are incorporated into the event scheduling mechanism. This is a crude approximation of the real processing delays experienced on a system and is application specific: different applications may have very different processing delay models. For example, our processing delay model would be inadequate for an application that reads and writes data one percent of the time and spends the remaining time performing expensive operations. Inaccurate delay models would potentially skew results. Therefore, new application performance testing is required to appropriately model processing delays. Further, experiments that attempt to introduce variability in processing times as a key feature of an algorithm or as part of an attack will require more accurate simulations.

In our analysis of Tor performance, our modeling approaches were suitable to obtain realistic and consistent results. In particular, our Tor experiments were run using the same models while each experiment varied only a single configuration option. Therefore, each experiment was subject to the same system and network conditions and the same systematic biases that Shadow potentially introduces during simulation. As a result, we can be reasonably convinced that the *relative* differences between experimental results are due to our configured change and not to some systematic inaccuracy inherent to simulation, irrespective of how well the *absolute* simulation results match those obtained from real systems and networks.

Future Shadow users should be aware of the above limitations and recognize that adequate models are application dependent. It may be the case that more robust models are needed to effectively analyze particular algorithms or protocols. Future research should adapt models as appropriate to their work in order to draw meaningful conclusions, and validate results with other experimentation methods.

3.7 Summary

This chapter presented the design and implementation of a large scale discrete event simulator called Shadow, and a plug-in called Scallion that is capable of linking to and running the Tor software over a simulated network. In addition to an explanation of Shadow's non-trivial design, we performed an extensive experimental analysis to verify the accuracy of Tor simulations. We found that client performance for simulated Tor clients is surprisingly congruent to performance achieved through the live public Tor network. High accuracy is achieved by "shadowing" the Tor network, considering relay characteristics from a live Tor consensus and inter-node latency characteristics from PlanetLab ping measurements.

Chapter 4

Methodically Modeling the Tor Network

4.1 Introduction

Recall that Tor [11] is an anonymizing overlay network consisting of thousands of volunteer *relays* that provide forwarding services used by hundreds of thousands of *clients*. To protect their identity, clients encrypt their messages multiple times before source-routing them through a *circuit* of multiple relays. Each relay decrypts one layer of each message before forwarding it to the next-hop relay or destination *server* specified by the client. Without traffic analysis, the client and server are *unlinkable*: no single node on the communication path can link the messages sent by the client to those received by the server.

Tor is a distributed system containing a handful of authorities that assist in distributing a *consensus* of trusted relay information. This *directory* of relays informs clients about the stability of and resources provided by each relay. Clients use this information to select relays for their circuits: the choice is weighted by the relative difference in the perceived throughput of each relay in an attempt to balance network load. Although Tor's main purpose is to protect clients' communication privacy, it also serves as a tool to resist censorship. Citizens in countries controlled by repressive regimes rely on Tor to mask their intended communication partners, thereby circumventing the block that may otherwise occur at censors' borders. Although several nations have attempted to block Tor, its distributed architecture has thus far proven resilient to long term censorship.

Tor's popularity, distributed architecture, and privacy requirements increase the difficulty in experimenting with new algorithm and protocol designs. New designs require software updates before testing their network effects, which both prolongs and complicates the experimental process. Further, since the live network is not a controlled environment, fluctuations in network conditions may both bias results and make them impossible to replicate. Finally, experiments are often constrained due to privacy risks to Tor's clients.

The disadvantages to live Tor experimentation have led researchers to explore alternative approaches, including the utilization of network testbeds such as PlanetLab [61], simulation [49, 46, 62], and emulation [63, 64, 65]. Each of these alternatives to experimentation on the live Tor network must make choices about how to model the existing network. A lack of details about and justifications for such choices obscures the level of faithfulness to the live network and decreases confidence that the obtained results provide meaningful information.

We improve the state of cyber security by contributing a novel and complete model of the Tor network that may be used for safe and realistic Tor experiments. In Section 4.3, we enumerate, explore, and justify each Tor modeling decision through methodical reasoning, using data from real Internet measurements where possible. We provide insight into non-intuitive consequences of alternative modeling strategies while precisely specifying and discussing our modeling techniques.

We validate that our model produces an accurate environment whose performance and load are characteristic of the live Tor network. To this end, we utilize two state-of-the-art Tor experimentation platforms: Shadow [62] and ExperimenTor [65]. We describe the tools and discuss their pros and cons in Section 4.2, both to show that our model is applicable in multiple testing environments and to guide future work in selecting the tool most suitable to a given research question. In Section 4.4, we instantiate our model with both Shadow and ExperimenTor and compare results obtained with each tool to data collected from the live Tor network. We find that both tools produce reasonable Tor load and performance characteristics using networks of various sizes produced with our model. We inform the research community about the lessons we learned in Section 4.5 while concluding in Section 4.6.

The following summarizes this chapter's contributions:

- We justify and precisely specify the techniques we used to create accurate Tor network models
- We validate that our model is capable of producing an accurate environment whose performance and load are characteristic of the live Tor network, using multiple experimentation tools
- We provide the first direct comparison between results obtained with Shadow [62] and ExperimenTor [65]—two state-of-the-art Tor experimentation platforms

4.2 Background

While Tor is the most widely used anonymity network today with hundreds of thousands of daily users, Tor is still an active research network on which researchers work to improve its performance and security. To that end, prior Tor research has utilized a wide variety of methodologies which includes: analytic modeling [49, 32] and simulation [49, 46] of specific aspects of Tor's design; relatively small Tor deployments on PlanetLab [41, 33]; and direct experimentation [66] and measurement [28] on the live Tor network.

Analytic modeling, simulations and small-scale Tor deployments on testbeds such as PlanetLab each make certain simplifying and potentially unrealistic assumptions that often leave many open questions about how the results obtained might translate to the live Tor network. Direct measurement and experimentation with the live network are unable to investigate design changes at scale due to software upgrade delays. Further, such well-intentioned research might have a negative impact on real Tor users' quality of service as well as their privacy [67].[1]

In an effort to enhance the realism and safety of Tor experimentation, two designs for whole-Tor network testbeds, Shadow [62] and ExperimenTor [65], have been independently developed and made publicly available for use by the research community. In contrast to prior approaches to Tor research, these testbeds seek to replicate in isolation the important dynamics of the live Tor network at or near scale, complete with directory authorities, Tor routers, Tor clients, applications, and servers. While the details of how these tools model the live Tor network are discussed at length in Section 4.3, we first overview each tool's distinct approach.[2]

4.2.1 Shadow

To produce high fidelity experiments in a controlled and repeatable manner, Shadow leverages discrete-event simulation of the network layer and runs real, unmodified application software within the virtual network topology. Shadow also simulates the effects of background Internet traffic by introducing non-deterministic jitter and packet loss on links. Shadow offers an extensible plug-in framework through which an investigator can integrate an application or protocol of her choice into the Shadow experimentation environment. A plug-in called Scallion for simulating the Tor network is available. Important advantages of Shadow are that it can simulate large-scale distributed systems

[1]See Bauer *et al.* [65] for a survey of prior methods for Tor research.

[2]This work describes and uses Shadow version 1.4.0 with Scallion version 1.3.1, and ExperimenTor as of April 2012. Later versions may have new features and capabilities not described here.

(such as Tor) on a single well-provisioned machine, results can be trivially replicated due to its design, and it can scale to arbitrarily sized networks because it runs in virtual time. Furthermore, virtual machines are available for running Shadow in the cloud on Amazon's EC2. See Shadow's webpage for more details [52].

4.2.2 ExperimenTor

Similar to Shadow, ExperimenTor offers the ability to run unmodified Tor software within an isolated environment to conduct experiments that are faithful to dynamics of the live Tor network. In contrast to Shadow's network simulation approach, ExperimenTor is a network emulation-based testbed, built on the mature Modelnet [68] network emulation platform. ExperimenTor uses one machine to emulate a specified network topology and another machine (or possibly several machines) to run unmodified software within the virtual network. Also unlike Shadow, ExperimenTor does not endeavor to account for the effects of unrelated background Internet traffic on experiments. While ExperimenTor cannot easily be run on a single machine, it has an advantage of using the operating system's native network stack, rather than a simulation. More details about ExperimenTor can be found on its webpage [69].

4.3 Model

Tor experimentation outside of the live network benefits from accurate models of network characteristics and node behaviors. This section details our approach to modeling Tor while discussing alternative approaches and common pitfalls. Our model is not intended to be a complete set of *all* characteristics and behaviors one *could* model, but rather the subset that we found most important and most useful. Although tested with Shadow [62] and ExperimenTor [65], we intend the model to apply to a broad range of research problems.

4.3.1 Topology

We first consider the structure of our experimental network. Ideally, our network topology would replicate the Internet architecture, including all autonomous systems (ASes), core, backbone, and edge routers, and all links between them. Such a structure would

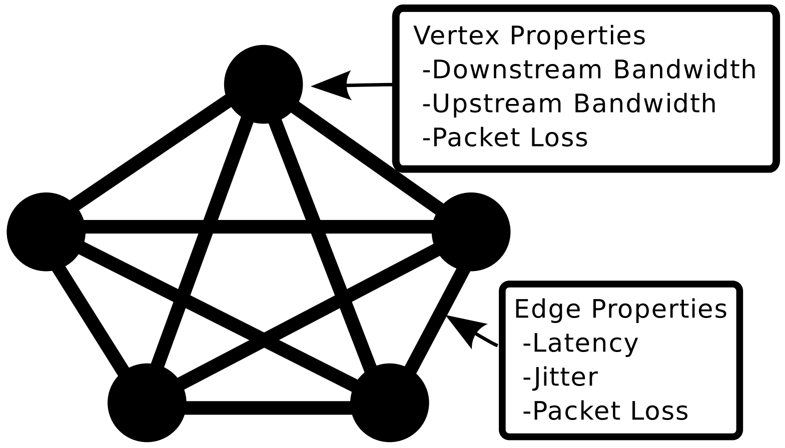


Figure 4.1: The vertex and edge properties in our modeled topology. The topology forms a complete graph.

provide the most accurate view of the Internet to an experimental framework. Unfortunately, the exact structure of the Internet is unknown and inferring it is an open research problem (e.g., [70]). Even if the Internet structure were known, it would be extremely large and too inefficient to replicate for experimental purposes. Therefore, we produce a small-scale, manageable model of the Internet.

Mapping the Internet topology is a major research area that has resulted in the development of multiple tools and techniques [71, 72, 73, 74]. This work utilizes geographic clustering *by country*[3] to scale the Internet down to a manageable topology because Tor similarly reports statistics about its users, allowing for a natural assignment of Tor node properties and placement of Tor nodes in our topology. Further, our approach produces small and efficient complete topologies (see Figure 4.1): we minimize the number of topology vertices and edges while remaining compatible with Tor's reporting method, and do not require routing algorithms to send packets through the network backbone. Finally, geographical clustering simplifies the process of mapping nodes to vertices, since any desired location (IP address) can be mapped to a cluster using a wide variety of GeoIP tools (e.g. those provided by MaxMind [60]).

[3]Note that some Tor research questions may require a more detailed model of the Internet topology, a problem future work should consider.

Network Vertices

In our clustering approach, we create a *network vertex* for each country, Canadian province, and American state.[4] We take this approach, as opposed to clustering by Autonomous System (AS), because geographical clustering most closely resembles the actual structure of the Internet: end-users and hardware are physically located in clearly defined geographical regions. ASes, however, typically span multiple geographical regions. Further, many properties of network vertices and edges directly correspond to their geographical location, resulting in less variance when aggregating measurements of such properties.

Each vertex is assigned default *upstream bandwidth*, *downstream bandwidth*, and *packet loss* properties obtained from the Ookla Net Index dataset [75]. The dataset provides aggregate statistics collected during bandwidth speed tests [76] and ping tests [77]. Ookla aggregates millions of such tests and provides the rolling mean throughput for each geographic region (vertex in our topology) over thirty day intervals. The cumulative distributions on bandwidths are shown in Figure 4.2(a).

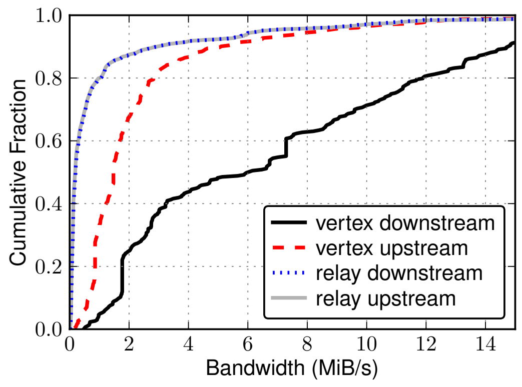
Network Edges

Each vertex in our topology is connected to every other vertex, forming a complete graph. Each of these pairwise connections are represented as a *network edge*. We assign each network edge the following properties: latency (end-to-end packet delay), jitter (the variation in packet delay), and packet loss (the fraction of packets that are dropped). Note that full end-to-end loss rates are computed by combining the loss rates of the source and destination vertices and the connecting edge. Due to the lack of accurate loss rate measurements in the Internet core, our model currently utilizes only vertex loss rates from Ookla [75].

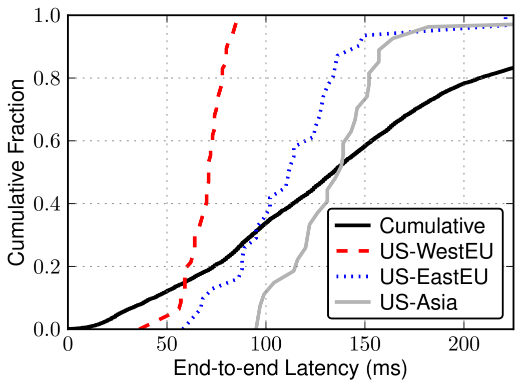
To model edge latency in our topology, we use round trip times (RTTs) measured by the iPlane [73] latency estimation service.[5] iPlane gathers RTTs from several vantage points, including PlanetLab nodes and traceroute servers, on a daily basis [78]. We use

[4]We used Tor's directly-connecting-user country database [30] to form our list of countries, which we supplemented with states and provinces from Net Index [75].

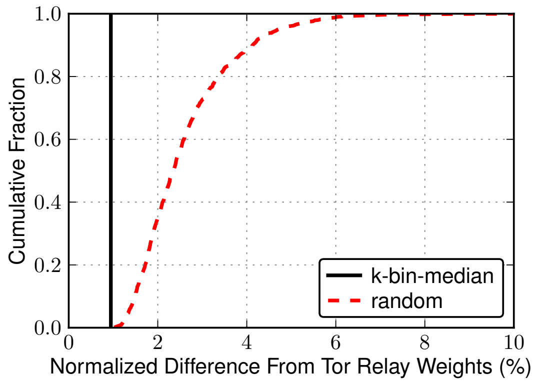
[5]The traceroutes were collected on 2012-03-28.



(a) Vertex Bandwidth



(b) Edge Latency



(c) Relay Sampling Accuracy

Figure 4.2: (a) Topology vertex bandwidths from Net Index, and estimated relay bandwidths from published relay documents. (b) Topology edge latency. (c) Sampling relays for scaled-down Tor experiments. Our sampling algorithm produces the best fit to the original relay bandwidth distribution by minimizing the area between the CDF curves.

$\frac{RTT}{2}$ to approximate latency between every iPlane node.[6] We then use GeoIP lookup to assign each iPlane node to a network vertex, and therefore each estimated latency value corresponds to a network edge. Since there may not be an iPlane node corresponding to every network vertex (because there is not an iPlane node in every country), we create a temporary virtual overlay topology containing only nine "regional" clusters (e.g. US East, US West, EU East, EU West, etc.) and aggregate our latency estimates on the corresponding regional overlay edges. Then, we assign each network edge for which we have no RTT measurements the median latency value from the corresponding overlay edge. Figure 4.2(b) shows the iPlane latency estimates between common regional overlay edges, and confirms that an increase in physical distance between nodes implies an increase in latency. Finally, we approximate jitter over our network edges as $\frac{IQR}{2}$, where IQR is the edge latency inter-quartile range.

4.3.2 Hosts

Once we have configured a topology, we next configure hosts that operate in that topology. In the context of a Tor network, we are most concerned with Tor relays, Tor clients, Tor authorities, and Internet web/file servers. Although the live Tor network contains thousands of relays and hundreds of thousands of clients, it is often the case that experiments must be scaled down significantly due to hardware limitations. We now explain our approach to scaling down the Tor network for each host type.

Tor Relays

Relays are an important part of a Tor network model, as each of them donates bandwidth and provides the forwarding service upon which the network is built. Recall that when building circuits, clients select relays according to weights published in the consensus. These selection-weights direct clients to relays according to each relay's perceived throughput, and have a dramatic impact on relay and network load and congestion [43]. Therefore when scaling down from the thousands of relays in the Tor consensus to a manageable number for experiments, it is important that the distribution of selection-weights in the scaled network is as close as possible to that of the live network. Past

[6]Although Internet paths may be asymmetric, we found $\frac{RTT}{2}$ a suitable approximation of edge latency after aggregating measurements.

Algorithm 1: Sample relay bandwidths to produce a distribution that best fits that of the original relay population

```
  Input: sorted list $\mathcal{L}$ of $\mathcal{N}$ relay bandwidths, sample size $\mathcal{K} \leq \mathcal{N}$
  Output: sorted list of sampled bandwidths $\mathcal{S}$
1 $n \leftarrow floor\left(\frac{\mathcal{N}}{\mathcal{K}}\right)$;
2 $r \leftarrow \mathcal{K} - n$;
3 $i \leftarrow 0$;
4 for $k \leftarrow 0$ to $\mathcal{K} - 1$ do
5     $j \leftarrow i + n$;
6     if $k < r$ then $j \leftarrow j + 1$;
7     $bin \leftarrow \mathcal{L}.slice(i, j)$;                    // range $[i, j)$
8     $\mathcal{S}.add(median(bin))$;
9     $i \leftarrow j$
```

work has sampled uniformly at random from the existing set of relays [46] when choosing relays for experiments. Unfortunately, the distribution that results from randomly selecting relays may not fit the original selection-weight distribution well. We now describe an algorithm that produces the *best fit sample* of the original distribution while quantifying its improvement over random selection.

To scale the number of relays down to $\mathcal{K}$ of $\mathcal{N}$, we split a sorted list of $\mathcal{N}$ relay selection-weights into $\mathcal{K}$ bins and choose the median weight from each bin. The resulting weight distribution *best fits* that of the original relay population: any non-median weight value would only increase the distance between the distributions. This approach is detailed in Algorithm 1. To quantify our algorithm's effectiveness, we compare it to random selection using the difference from the original relay weight distribution as a metric. This is calculated as the integral of the absolute value of the difference between the sampled CDF $s(x)$ and the Tor relay selection-weight CDF $f(x)$:

$$\int_0^\infty \mid f(x) - s(x) \mid dx \tag{4.1}$$

The result is then normalized. Figure 4.2(c) compares the distribution of this closeness metric for 1000 samples of $\mathcal{K}$ relays using our algorithm and random sampling.[7] While our algorithm always produces the best fit result for each sample (the vertical line

[7]We found insignificant variance in the sample distributions when choosing $\mathcal{K} \in [50, 1000]$.

Algorithm 2: Estimate relay upstream and downstream capacities using data published in the consensus, server descriptors, and extra infos

```
Input: consensus weights $\mathcal{C}$, max bw bursts $\mathcal{B}$, max read and write bw histories
       $\mathcal{R}$ and $\mathcal{W}$
Output: capacity up $\mathcal{U}$ and down $\mathcal{D}$
1  for $i \leftarrow 0$ to $getRelayCount() - 1$ do
2      if $\mathcal{B}[i] > 0$ then
3          if $\mathcal{R}[i] > 0$ and $\mathcal{W}[i] > 0$ then
4              $ratio \leftarrow \frac{\mathcal{R}[i]}{\mathcal{W}[i]}$;
5              if $ratio > 1$ then
6                  $\mathcal{U}[i] \leftarrow \mathcal{B}[i]$;
7                  $\mathcal{D}[i] \leftarrow (\mathcal{B}[i] \cdot ratio)$;
8              else
9                  $\mathcal{D}[i] \leftarrow \mathcal{B}[i]$;
10                 $\mathcal{U}[i] \leftarrow \left(\mathcal{B}[i] \cdot \frac{1}{ratio}\right)$;
11         else $\mathcal{U}[i] \leftarrow \mathcal{D}[i] \leftarrow \mathcal{B}[i]$;
12     else if $\mathcal{R}[i] > 0$ and $\mathcal{W}[i] > 0$ then
13         $\mathcal{U}[i] \leftarrow \mathcal{W}[i]$;
14         $\mathcal{D}[i] \leftarrow \mathcal{R}[i]$;
15     else $\mathcal{U}[i] \leftarrow \mathcal{D}[i] \leftarrow \mathcal{C}[i]$;
```

in Figure 4.2(c)), random sampling produces distributions as far as ten percent from optimal.

We draw two samples of relays from those listed in the consensus: one for exit relays (discussed below) and one for non-exit relays. We then consider several relay properties. First, we assign each relay to the network vertex in our topology corresponding to its geographic location (found by GeoIP lookup of the relay's IP address). This allows communication between relays while also resulting in latencies between relays that correlate with physical distances. Next we compute relay rate limits[8] and access link capacities, the most important properties affecting the resources each relay provides and the expected client performance in our modeled network. Rate limits are taken from the public server descriptors [30] of our sampled relays. Capacities must be estimated.

[8]A relay operator may limit the amount of bandwidth its relay consumes by configuring a token bucket rate-limiter: the token bucket size and refill rate can be configured by setting `BandwidthBurst` and `BandwidthRate` in the configuration file.

Since a relay's ISP access link capacities are not directly measured or published, we estimate these values using historical bandwidth measurements published in server descriptors and extra info documents, and the weights published in the consensus. The published documents include: bandwidth weights—values used during circuit construction to help distribute client load to faster relays; observed bandwidth—the smaller of the maximum sustained input and output over any ten second interval; and read/write bandwidth histories—the maximum sustained input and output over any fifteen-minute interval. We prefer the observed bandwidth as the best estimate of capacity. Since only the smaller of the input and output observed bandwidth is published, we use the read/write histories to infer to which the published value corresponds, and the ratio of read/write histories to estimate the unpublished observed value. In the absence of observed bandwidth information, we use read/write histories directly, and otherwise fall back on the bandwidth weights. A detailed specification is provided in Algorithm 2. The distribution on relay bandwidths computed using Algorithm 2 is shown in Figure 4.2(a).

Note that a relay's observed bandwidth is only a good estimator of capacity when the relay was not limiting its rate during at least one ten second interval, and the relay had enough clients to consume its available bandwidth. Otherwise, the observed bandwidth is an underestimate of a relay's true capacity. This is corroborated in Figure 4.2(a): upstream and downstream estimates are mostly symmetric due to the reliance on observed Tor bandwidth and Tor's circuit design, and the relay capacities appear far less than the expected upstream and downstream capacities from Net Index. We plan to explore passive measurement techniques, such as packet trains [79], to directly measure relay capacities in future work. Such measurements would provide a significantly better data source for modeling capacities than currently available.

The last part of modeling relays is adjusting their Tor configuration. As mentioned above, we sample relays that will exit Tor traffic separate from those that won't. Both exit and non-exit relays require the `ORPort` option to configure it as a relay while exit relays additionally require a configured `ExitPolicy`. (Exit policies may be found in relays' server descriptors.) Other notable configurations include `TestingTorNetwork` to help with bootstrapping in our test environment, and `DirServer` to specify our directories.

Tor Authorities

Tor *directory authorities* are responsible for creating, signing, and distributing the consensus document—a list of all available relays and their associated bandwidth weights. Tor *bandwidth authorities* measure the expected performance of each relay and use the relative measured performance to compute the consensus weights used by clients for relay selection. In the live Tor network, the bandwidth measurement functionality is provided by a set of scripts known as TorFlow [57].

Our model selects the fastest sampled non-exit relay as the directory authority (all Tor directory authorities are currently non-exit relays). Since our test network lacks TorFlow, we must ensure that the bandwidth weights that appeared in the live network consensus also appear in our test network consensus. This is done by writing a `.v3bw` file with the live network bandwidth weights in the directory authority's data directory, as is done in live Tor. Lacking a valid `.v3bw` bandwidth file, the authorities will fall back on relays' reported observed bandwidth. In this case, we must remove a software-defined limit[9] on the observed bandwidth to allow relays to report the correct consensus weight. Note that although clients will be selecting relays in our test network using the same weights as the live network, the probability that each relay is selected necessarily increases (we downsampled the relays and the sum of the probabilities must equal 1).

Tor Clients

In our model, Tor clients are the main source of network load, producing all of the exit-bound traffic routed through Tor while simultaneously serving to measure network performance. Clients perform synchronous `HTTP GET` requests to download files through our modeled Tor network. Clients choose `HTTP` servers from which to request each download uniformly at random. Since the requests are synchronous, each client will be responsible for at most one stream through Tor at any time. Each client measures the time from when it initiates a connection to the SOCKS application proxy to the first byte and last byte of the file payload, indicating network responsiveness and performance.

Our model classifies clients into two broad categories: web clients and bulk clients. Each web client requests 320 KiB files, the average webpage size according to recent web

[9]Directory authorities will not trust any self-reported relay bandwidth over `DEFAULT_MAX_BELIEVABLE_BANDWIDTH`, which is set to a default value of 10 MiB/s.

Table 4.1: The ten countries with the highest reported Tor connecting user counts [30] during January, 2012.

Country	%	Country	%
United States	16.46	Spain	5.08
Iran	12.63	Russia	3.46
Germany	9.99	Republic of Korea	2.66
Italy	6.96	United Kingdom	2.39
France	6.30	Saudi Arabia	2.38

metrics [80]. After completing a download, a web client will pause for a time drawn uniformly at random from a range of $[1, 20]$ seconds before initiating the next download to simulate the time a user takes to consume the web page content. Each bulk clients requests 5 MiB files without pausing between the completion of one download and the initiation of the next. Our client model is based on work characterizing Tor exit traffic by McCoy *et al.* [28]. This work found that roughly 60% of the bytes and 95% of the connections exiting Tor were attributable to `HTTP` traffic while roughly 40% of the bytes and 5% of the connections were attributable to BitTorrent traffic. Therefore, we use a 19:1 *ratio* of web to bulk clients. The total *number* of clients is dependent on the number of relays and their capacities (see Section 4.4).

Each client is assigned a geographical location and the corresponding network vertex in our topology according to Tor's directly connecting user statistics [30, 81]. These statistics specify the country from which clients connect when directly downloading Tor directory information. The top ten countries from a recent version of this data are shown in Table 4.1. When assigning a client to a vertex, the assignment is weighted by the given percentages. Each client's upstream and downstream capacities on the connection to and from its ISP are taken from the default vertex properties as measured by Net Index [75] (see Section 4.3.1).

Internet Servers

In our model, `HTTP` servers are the destinations of our client requests and the sources of the files downloaded through Tor. In order to attribute changes in performance to Tor itself while minimizing effects external to the network, we assign Internet servers

Table 4.2: The ten countries with the highest number of servers in the Alexa top 1 million data set [82] during January, 2012.

Country	%	Country	%
United States	47.94	France	3.64
Germany	8.65	Russia	3.40
China	4.50	Netherlands	2.86
United Kingdom	4.20	Canada	2.10
Japan	3.73	Italy	1.48

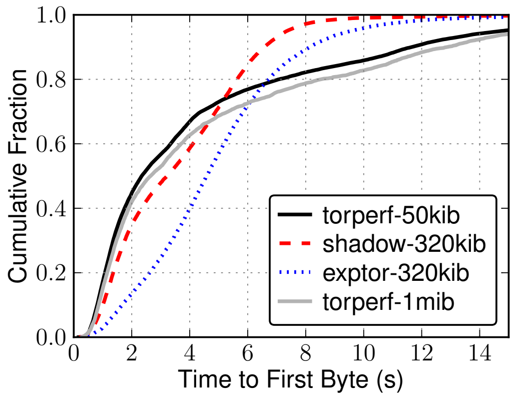
100 MiB/s bandwidth capacities. This high capacity will prevent our Internet servers from becoming bottlenecks during our client downloads. The geographic locations of Internet servers are assigned using the Alexa Top Sites data set [82]. Since the Alexa *ranking* may not capture the usage patterns of Tor users well, we instead produce a distribution on location of the reported top one million sites.[10] The top ten countries with the most sites in the Alexa data set are given in Table 4.2. Our assignment of server to topology vertex is weighted by this distribution, similar to our client vertex assignment.

4.4 Methodology and Experiments

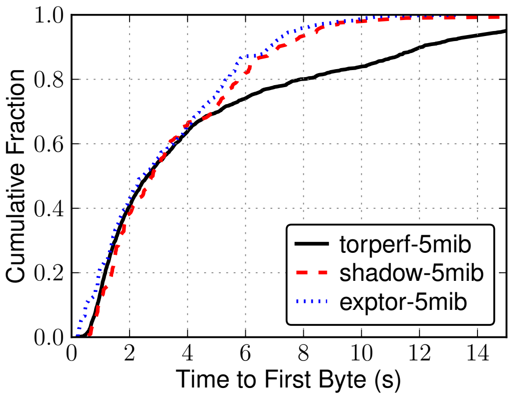
To determine the accuracy of and increase the confidence in our Tor network model, we instantiate it using two state-of-the-art Tor experimentation tools: Shadow [62] and ExperimenTor [65] (see Section 4.2 for background). This section compares the performance and load characteristics of the environments produced with each tool to that of the live Tor network, illustrating the effectiveness of our modeling strategies from Section 4.3. We choose network performance and load because Tor already measures these characteristics on the live network, allowing for a direct comparison of results. Further, these metrics represent the gauges in which clients and relays are generally interested, and are most useful when developing new algorithms that improve the state of the network.

We test our model with two different network sizes, both of which are scaled down

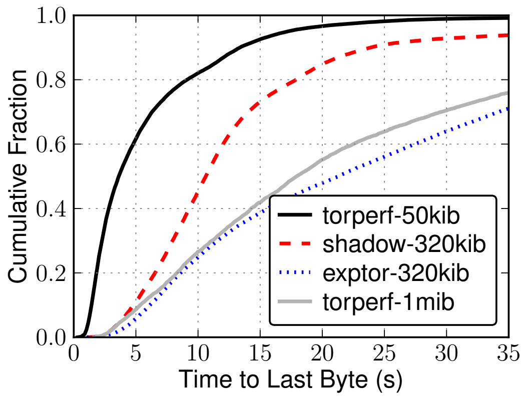
[10]We find locations with standard DNS queries and GeoIP lookups.



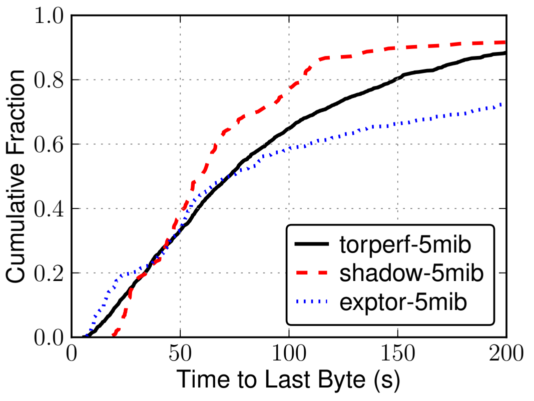
(a) 320 KiB, First Byte



(b) 5 MiB, First Byte

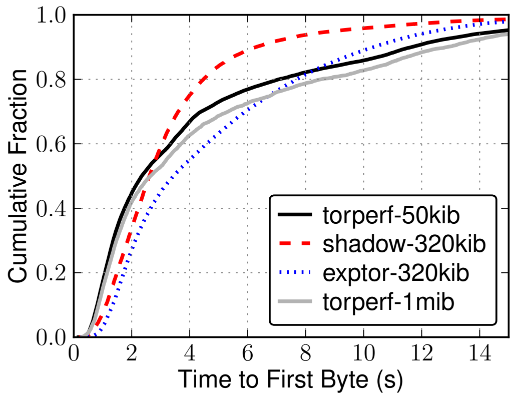


(c) 320 KiB, Last Byte

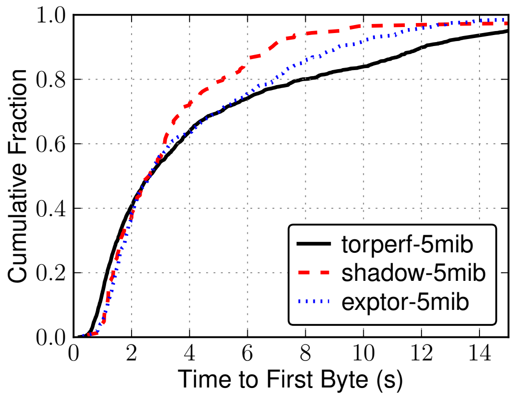


(d) 5 MiB, Last Byte

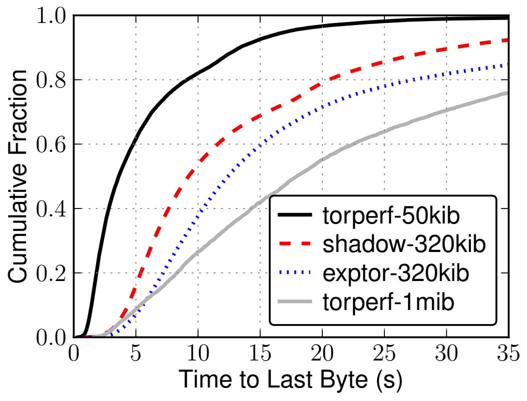
Figure 4.3: Performance for live Tor and our small modeled network configured with 50 relays and 500 clients in Shadow and ExperimenTor. Time to the first byte of the data payload is shown in (a) and (b), and time to the last byte in (c) and (d), for various download sizes.



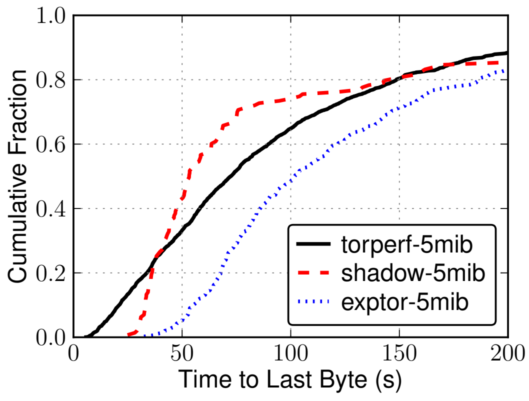
(a) 320 KiB, First Byte



(b) 5 MiB, First Byte



(c) 320 KiB, Last Byte



(d) 5 MiB, Last Byte

Figure 4.4: Performance for live Tor and our large modeled network configured with 100 relays and 1000 clients in Shadow and ExperimenTor. Time to the first byte of the data payload is shown in (a) and (b), and time to the last byte in (c) and (d), for various download sizes.

versions of the live Tor network. In our *small* network, we configure 50 relays and 500 clients that communicate with 50 HTTP file servers. In our *large* network, we configure 100 relays and 1000 clients that communicate with 100 HTTP file servers. The small and large networks are approximately fifty and twenty-five times smaller than the size of Tor, respectively. Both Shadow and ExperimenTor use instantiated versions of our Tor network model[11] and are configured to run a vanilla instance of version 0.2.3.13-alpha of the Tor software for ninety virtual minutes. Download results are ignored during the first thirty minutes of each experiment to allow for Tor's bootstrapping process. File download timings during the remaining period are utilized as discussed below.

Note that we explored various numbers of clients and found that a 10:1 client-to-relay ratio in our experiments resulted in load and network performance that reasonably approximated that of the live Tor network [30]. We stress that this client-to-relay ratio is due to our client modeling strategies; alternative client behaviors may require an adjusted ratio to produce the network characteristics that best approximate Tor. Accurately modeling Tor client behaviors is an open research problem which future work should consider.

4.4.1 Network Performance

We compare client performance measured in our test environments to client performance in Tor during the same period we are modeling.[12] We measure the time to the first and last byte of the data payload of our 320 KiB and 5 MiB file downloads as indications of network responsiveness and throughput. We compare our results to live Tor network performance measured with torperf [83], a tool that monitors live Tor network performance by downloading files of sizes 50 KiB, 1 MiB, and 5 MiB. Performance for our small and large networks are respectively shown in Figures 4.3 and 4.4.

We expect client performance in our test environments to be similar to that in Tor. In particular, the time-to-first-byte should be consistent regardless of the size of the file being downloaded. As can be seen in Figures 4.3(a), 4.3(b), 4.4(a), and 4.4(b), our model produces accurate time-to-first-byte performance in both tools, although the

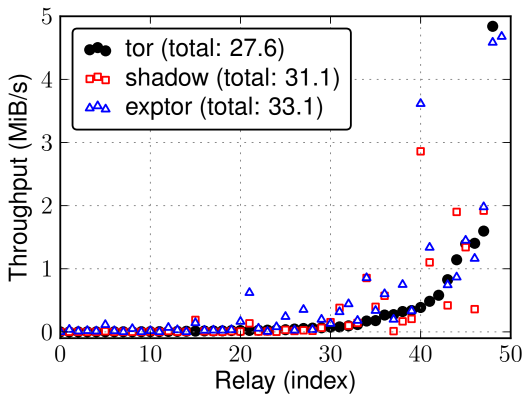
[11]The topology files are available on the Shadow website [52].
[12]This work models Tor as it existed during January, 2012.

tools tend to lose some accuracy above the eightieth percentile. Under the time-to-last-byte metric, we expect our 320 KiB web downloads to complete somewhere between the torperf 50 KiB and 1 MiB downloads, and our 5 MiB download times to be consistent with torperf. Web download times are more accurate in ExperimenTor in the large network (Figure 4.4(c)) than the small (Figure 4.3(c)), and all downloads tend to take slightly longer in ExperimenTor than in live Tor. Shadow approximates web download times reasonably well (Figures 4.3(c) and 4.4(c)), and bulk downloads complete slightly faster in Shadow than in Tor (Figures 4.3(d) and 4.4(d)). Overall, we are impressed that our model enables both tools to characterize Tor performance closely, even with scaled-down Tor networks.

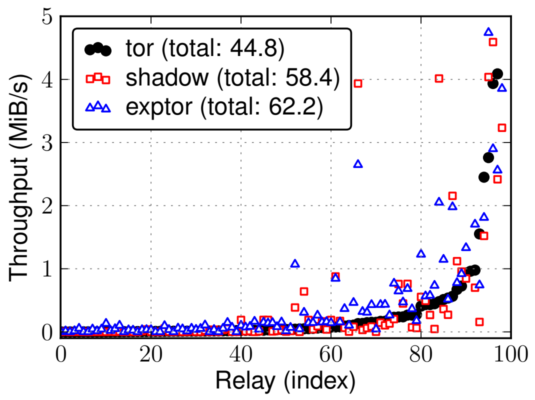
4.4.2 Network Load

Each relay in Tor tracks byte histories: the number of bytes read and written over time. We use these statistics to calculate the throughput of each relay included in our small and large networks, and directly compare throughputs from Tor with throughputs from our experimentation environments. The results are shown in Figure 4.5.

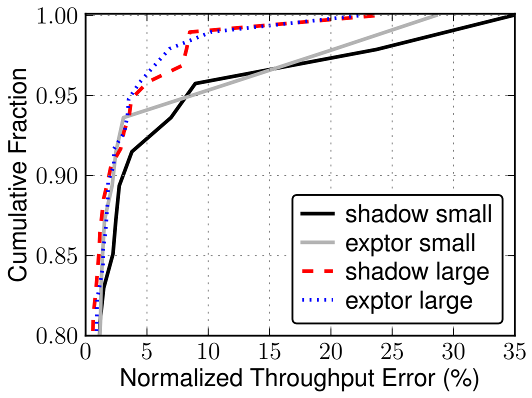
The aggregate throughput for all the relays we chose in our small network (Figure 4.5(a)) totaled 27.6 MiB/s for live Tor, 31.1 MiB/s in Shadow, and 33.1 MiB/s in ExperimenTor. In our large network (Figure 4.5(b)), the aggregate throughput was 44.8 MiB/s in live Tor, 58.4 MiB/s in Shadow, and 62.2 MiB/s in ExperimenTor. These results indicate that our experimental networks were too heavily loaded, and the absolute error increased with the network size. The distribution on the normalized individual relay throughput error is shown in Figure 4.5(c). The distributions have long tails: the maximum normalized error was 34.9% for Shadow and 28.6% for ExperimenTor in the small network, and 23.9% for Shadow and 22.5% for ExperimenTor in the large network. Although the absolute error increased with the network size, the individual errors decreased in the larger network. Our analysis found that most throughput error was attributable to bootstrapping issues: recently added fast relays were under-utilized in Tor but fully utilized in our experiments. Despite these issues, over 95% of the relays in the small network and 98% of the relays in the large network had less than 10% throughput error.



(a) Small network



(b) Large network



(c) Error comparison

Figure 4.5: Load in live Tor and (a) our small modeled network of 50 relays and 500 clients, and (b) our large modeled network of 100 relays and 1000 clients. Throughput is indexed by each relay chosen in our model and the sum is shown in the legend. (c) The distribution on the normalized experimental throughput error from reported live Tor throughput.

4.5 Lessons Learned

Modeling a distributed system is a complex process. During this process, we found that it is important to use real Internet and system measurements to eliminate arbitrary modeling decisions, as this tends to have a significant impact on how accurately the experimental environment replicates the real distributed system. However, measurements should not be used until they are fully understood (what they mean and how they are useful), or they may harm accuracy.

We also found it important to determine useful metrics that allow for a comparison between the experimental platform and the real distributed system being modeled. Useful metrics and proper comparisons of measurements increase confidence in the obtained results. Useful metrics assist in understanding the strengths and weaknesses of a model, and help determine if the environment produced from the model is suitable for the research question of interest.

We discovered that it's very useful to replicate experiments on multiple experimental platforms. This can help identify errors or peculiarities caused by a specific tool. For example, this process allowed us to discover that packet header overhead on TCP packets without a data payload were not consuming bandwidth on Shadow's virtual network interfaces. Shadow's accuracy improved greatly after accounting for TCP packet header overhead on both data and control packets.

Finally, we'd like to stress the importance of understanding that ExperimenTor and Shadow have fundamentally different approaches to experimentation: Shadow simulates all network properties including jitter and packet loss on links due to the presence of background Internet traffic; and ExperimenTor emulates link properties simply as a function of the Tor traffic load, ignoring any effects due to background Internet traffic. As in our experiments, differences between other tools may also contribute to the differences in experimental performance and load, and should be considered when analyzing results.

4.6 Summary

This chapter explored modeling the distributed Tor network. We provided precise and detailed specifications of our modeling choices and their effect on the resulting experimental environment. We validated our model by instantiating it in two state-of-the-art Tor experimentation tools: Shadow[62] and ExperimenTor[65]. We compared network performance and network load from our experiments to real Tor data and found that our model leads to environments that characterize the live Tor network well. Finally, we provided insights into the lessons we learned while replicating our experiments with multiple tools.

Chapter 5

Prioritized Tor Circuit Scheduling

5.1 Introduction

Recall that Tor supports hundreds of thousands of Tor clients that route data through only a few thousand bandwidth-limited Tor relays. The total demand for resources from such a large set of clients far outweighs the supply required for the network to operate efficiently. The high load clients place on the network increases the effect of network bottlenecks and results in congestion and performance problems. These problems inherently reduce Tor's usability, limiting both its scalability and its ability to support a larger and more diverse set of clients. Since Tor's usability correlates with its performance, a slower network leads to a smaller and less diverse network that also provides weaker anonymity to its users [25].

Tor's high network utilization has offered a unique opportunity to investigate its protocol and algorithmic inefficiencies [24]. One area of focus has been on Tor's circuit scheduling algorithm, which is used by relays to determine the order in which client-constructed circuits are allowed to send data when there are resources available for doing so. Tor's original design used a traditional round-robin scheduler [84, 85], however, analysis of Tor client traffic [28] provided a key insight that fueled the development of an alternative scheduling strategy. The study found that BitTorrent file sharing accounted for roughly forty percent of the traffic Tor transfers, but only about five percent of the connections. Under a fair round-robin scheduler, this small set of high bandwidth clients were consuming an unfair share of network resources.

Recognizing that there are different types of clients with different performance requirements, Tang and Goldberg introduced a new circuit scheduler that prioritizes circuits with the lowest exponentially-weighted moving average (EWMA) throughput [33]. Their goal was to improve performance for latency-sensitive web clients with bursty traffic characteristics without reducing the long-term throughput of bulk traffic clients. They performed small-scale experiments that showed the new scheduler was able to improve performance for bursty traffic.

There are some shortcomings to the EWMA scheduler and its evaluation. First, the extent to which performance improves when all relays in the network are using the new scheduler is unclear, as Tang and Goldberg did not attempt experiments on a full-network deployment. Second, there is no way to control or specify the extent to which qualifying traffic should get prioritized: bulk clients whose traffic has been waiting long enough will be sent ahead of bursty traffic, even if this is not desired. Finally, the EWMA scheduler provides some notion of priority for bursty traffic, but does not allow performance adjustments between various other classes of traffic.

To overcome these shortcomings, this chapter first explores and builds upon the work of Tang and Goldberg: using Shadow, we investigate scheduling as a technique to improve client performance. In Section 5.2 we explore the EWMA circuit scheduler [33] which prioritizes bursty circuits ahead of bulk circuits. We confirm previous results by re-evaluating EWMA when enabled on *small* single-circuit topologies consisting of three relays – similar to those tested by Tang and Goldberg. However, our results from a *full-network* deployment of the scheduler in a scaled topology indicate that performance benefits are highly dependent on network load and a properly tuned half-life. We found that the scheduler *reduces* performance for Tor clients under certain network loads, a significant result since the EWMA scheduler is currently enabled by default for all sufficiently updated Tor relays.

Section 5.3 then explores the differentiated services architecture [86] in the context of the Tor network by evaluating two proportional differentiation [34] circuit schedulers – a *proportional delay differentiation* scheduler based on work by Dovrolis *et al.* [87] and a *proportional throughput differentiation* scheduler based on Tang and Goldberg's EWMA algorithm. We evaluate these schedulers using both single-circuit and full-network deployments in Shadow. We explore and discuss the fundamental extent to which scheduling may be used to improve Tor client performance: we evaluate our schedulers using both perfect classification as well as a simple classification heuristic. We find that scheduling can provide only modest performance improvements for Tor clients.

5.2 EWMA Circuit Scheduling

We now demonstrate Shadow's powerful capabilities by exploring a Tor circuit scheduling algorithm recently proposed and integrated into the Tor software.

5.2.1 EWMA Scheduling Model

In Tor, whenever there is room in an output buffer, the circuit scheduler must make a decision about which circuit to flush. Tor's original design used a round-robin algorithm for making such decisions. Recently, an algorithm based on the Exponentially-Weighted Moving Average (EWMA) of cells sent in each circuit was proposed and incorporated into Tor, and has since become the default scheduling algorithm used by Tor relays. This section attempts to validate the results originally obtained by Tang and Goldberg [33].

In order to prioritize bursty circuits, Tang and Goldberg's scheduler uses an EWMA throughput to represent "the number of cells a circuit has sent recently." This metric represents an average throughput of a circuit, but decays over time. This means that cells sent longer in the past will count less towards the EWMA metric than cells sent recently. The metric is computed by keeping a count n of the number of cells sent by a circuit. When a scheduling decision needs to be made, the EWMA scheduler chooses the circuit with the lowest cell count and updates n. The cell count n is decayed after time interval Δt using half life H (after time H, the count is reduced by half):

$$n' = n \cdot 0.5^{\frac{\Delta t}{H}} \tag{5.1}$$

Bursty traffic (web) should have lower EWMA cell counts and should be prioritized over steady traffic (bulk).

5.2.2 EWMA in Single-Circuit Topology

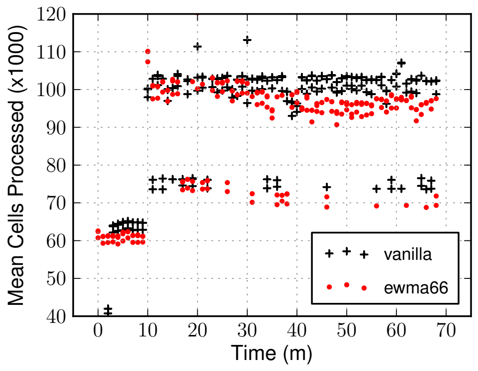
Tang and Goldberg evaluated the EWMA algorithm by creating a congested circuit on a synthetic PlanetLab network and measuring performance of web downloads. Since the middle node was a circuit bottleneck, the benefits of EWMA for reducing web download times were clear. Unfortunately, results for bulk downloads during this experiment were not given.

We perform a similar "bottleneck" experiment in Shadow. We configure a circuit consisting of a single entry, middle, and exit relay. Two bulk clients continuously download 5 MiB files to congest the circuit. Ten minutes after booting these "congestion" clients, two "measurement" clients are started and download for an hour: a third bulk client and a web client that waits 11 seconds (the median think-time for web browsers [58]) between 320 KiB file downloads. The middle relay is configured as a circuit bottleneck with a 1 MiB/s connection while all other nodes (relays, clients, and server) have 10 MiB/s connections.

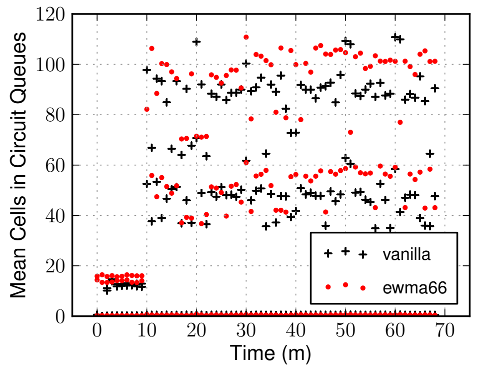
We run the above experiment modifying only the scheduling algorithm. We test both the round-robin scheduler and the EWMA scheduler with a Tor `CircuitPriorityHalflife` configuration of 66 as in [33]. Relay buffer statistics [88] are shown in Figures 5.1(a) and 5.1(b). Notice a significant increase in traffic at the ten-minute mark, at which point the "measurement" clients start downloading. Figure 5.1(a) shows that the number of processed cells is similar for all relays, except occasionally the exit relay processes fewer cells due to middle relay congestion. Figure 5.1(b) shows that the circuit queues increase for the exit and middle relay while the entry relay's circuit queues are empty due to sufficient bandwidth to immediately forward data to the client.

Figures 5.1(c) and 5.1(d) show the performance results obtained from the web client for both schedulers. As expected, the time to the first byte of the data payload and the time to complete a download are both reduced for the web client, since bursty traffic gets prioritized ahead of the bulk traffic. The time to first byte for the "measurement" bulk downloader in Figure 5.1(e) also improves for a large fraction of the downloads because each new download originating from a new circuit will be prioritized ahead of the "congestion" bulk downloads. However, after downloading enough data, the "measurement" bulk client loses its priority over the "congestion" bulk clients and the time to first byte converges for each scheduler.

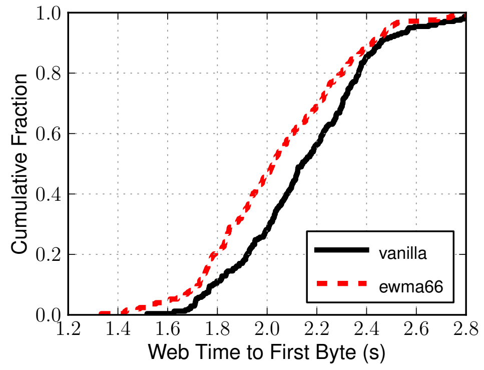
Tang and Goldberg claim that, according to Little's Law [89], bulk transfers will not be negatively affected while using the new circuit scheduler. While this may be theoretically true, it is not clear that it will hold in practice. The authors find that Little's Law holds when a single relay in the live Tor network uses the EWMA scheduler: their results show that bulk download times are not significantly different for each scheduler. However, our results in Figure 5.1(f) indicate otherwise. Bulk download times are noticeably worse for the EWMA scheduler, with a significant increase at around the 40th percentile. This increase again happens when the "measurement" bulk client loses its priority over the "congestion" bulk clients, suggesting that a deeper analysis of the EWMA scheduling algorithm in different network environments may be appropriate.



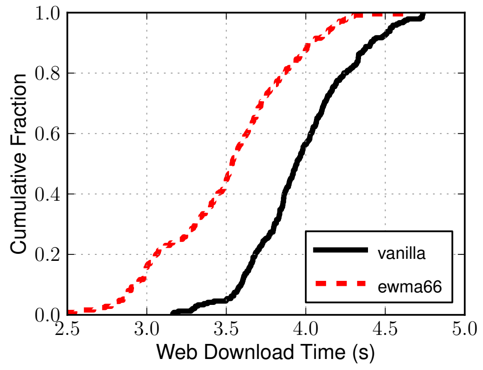
(a) Cells Processed



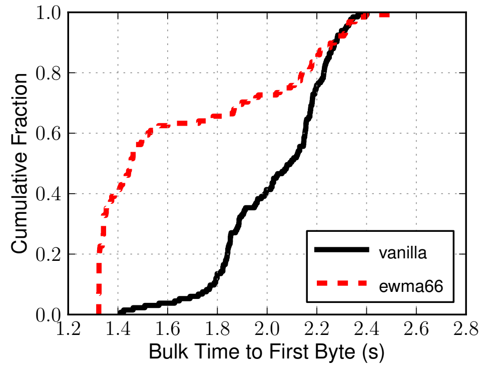
(b) Cells Queued



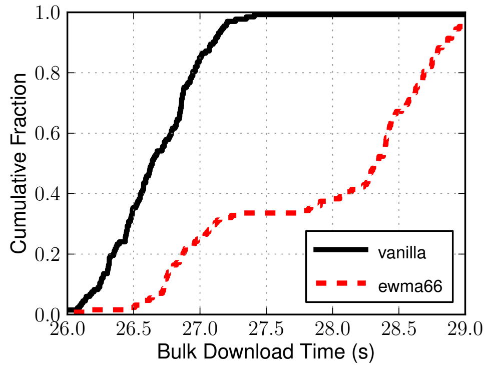
(c) 320 KiB, First Byte



(d) 320 KiB, Last Byte



(e) 5 MiB, First Byte



(f) 5 MiB, Last Byte

Figure 5.1: Seven-node single-circuit experiment similar to that performed by Tang and Goldberg [33]. The number of cells processed (a) and queued (b) increases at Time=10, when the measurement clients begin downloading. The EWMA scheduler improves responsiveness for bursty traffic (c), (d), and (e) but, contrary to the author's claims, decreases performance for bulk downloads (f).

5.2.3 EWMA in Full-Network Deployment

Tang and Goldberg's experiments suffer from a major limitation of scale: the experiments were run either on three-node PlanetLab topologies, or in the live Tor network with only a single relay scheduling with the EWMA algorithm. Although they provide results for what a single relay might expect when switching scheduling algorithms, they do not consider the network-wide effects of a full-network deployment.

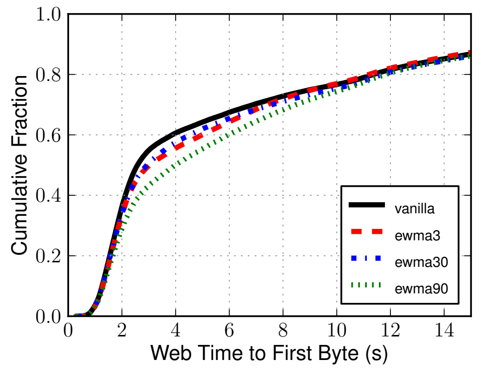
We explore the performance gains possible with the EWMA scheduler through a full network deployment in Shadow. We test the EWMA circuit scheduler with a range of half-life configurations and compare performance to the round-robin scheduler used in vanilla Tor. As in Section 3.5.3, we use 200 servers, 50 relays and 950 web clients for our experiments. To analyze the effects of various network loads on the scheduler, we run separate experiments configured with each of 25, 50, and 100 bulk clients. The adjusted load is significant since bulk clients account for a large fraction of network traffic. To reduce random variances, we run each experiment five times and show the cumulative results of each configuration by aggregating the results of all five experiments. Our results are shown in Figure 5.2 and Figure 5.3.

Under a "light" load of 25 bulk clients, Figures 5.2(a), 5.2(b), 5.3(a), and 5.3(b) show that the EWMA circuit scheduler reduces performance over vanilla Tor for all clients, independent of the configured half-life. Bulk download times seem to be affected the most (5.3(b)), but our experiments indicate there is also a significant reduction in responsiveness for web clients (5.2(a)).

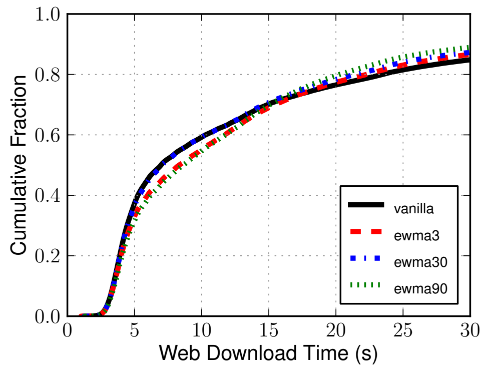
Under a "medium" load of 50 bulk clients, Figures 5.2(c), 5.2(d), 5.3(c), and 5.3(d) show that there are half-life configurations that still reduce performance when compared to vanilla Tor. The 30 and 90 second EWMA half-life configurations appear to improve performance for web clients (5.2(c) and 5.2(d)), but performance for bulk clients is either reduced or shows less improvement (5.3(c), 5.3(c)). Performance is reduced for all clients when using a 3 second half-life.

Finally, Figures 5.2(e), 5.2(f), 5.3(e), and 5.3(f) show performance under a "heavy" load of 100 bulk clients. Under heavy load, the EWMA scheduler appears to perform the best for web clients (5.2(e) and 5.2(f)) while bulk clients see no improvements over the vanilla Tor (round-robin) scheduler (5.3(e) and 5.3(f)).

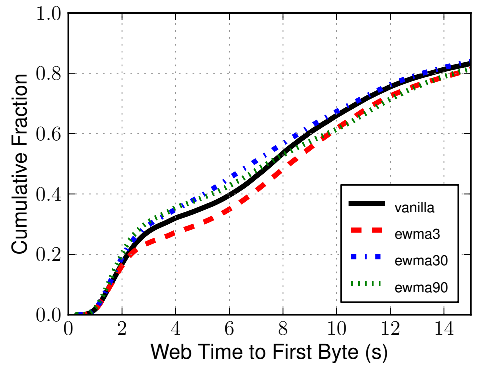
We conclude from our results that the EWMA scheduler should not necessarily be



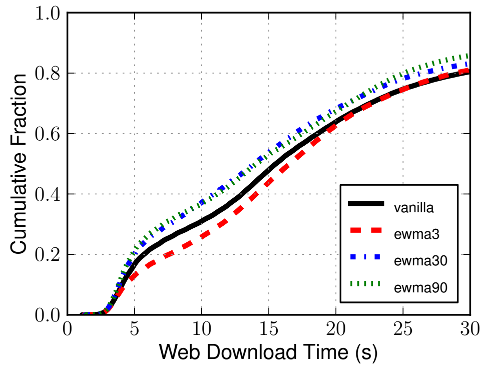
(a) 320 KiB, First Byte, Light Load



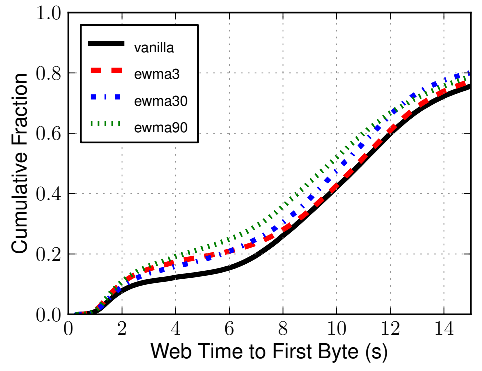
(b) 320 KiB, Last Byte, Light Load



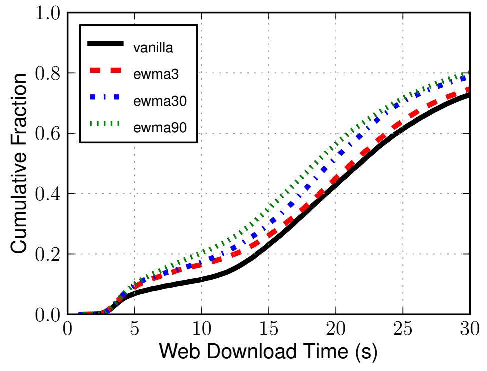
(c) 320 KiB, First Byte, Medium Load



(d) 320 KiB, Last Byte, Medium Load

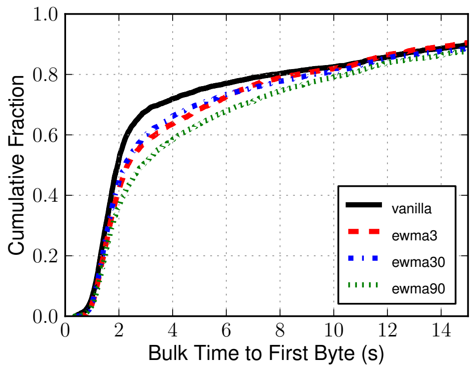


(e) 320 KiB, First Byte, Heavy Load

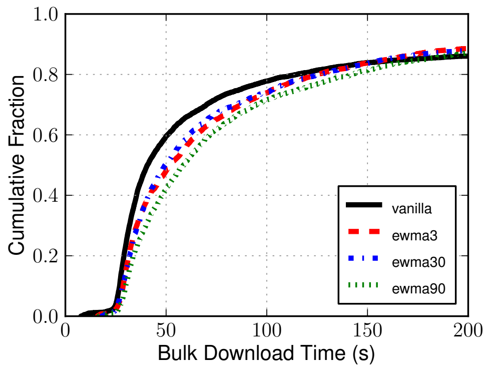


(f) 320 KiB, Last Byte, Heavy Load

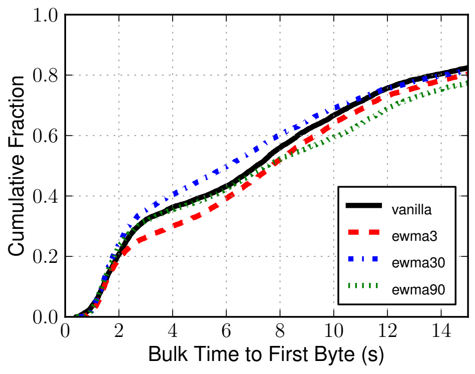
Figure 5.2: Performance comparison for the 950 web clients (320 KiB) under “light,” “medium,” and “heavy” loads of 25, 50, and 100 bulk clients, respectively, in full-network deployments using the EWMA circuit scheduler and the vanilla Tor round-robin circuit scheduler. While the EWMA circuit scheduler works best under heavily loaded networks, there are half-life configurations that reduce client performance.



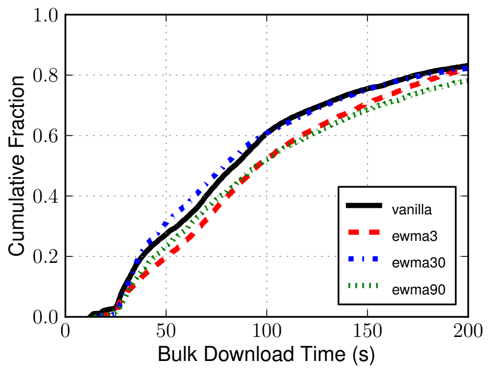
(a) 5 MiB, First Byte, Light Load



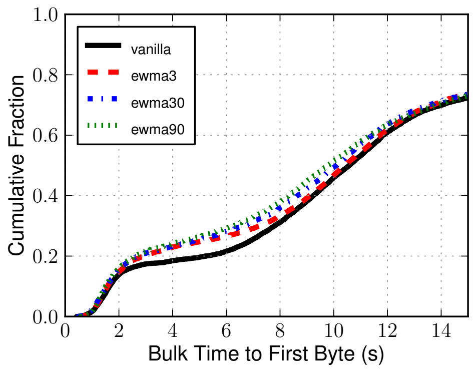
(b) 5 MiB, Last Byte, Light Load



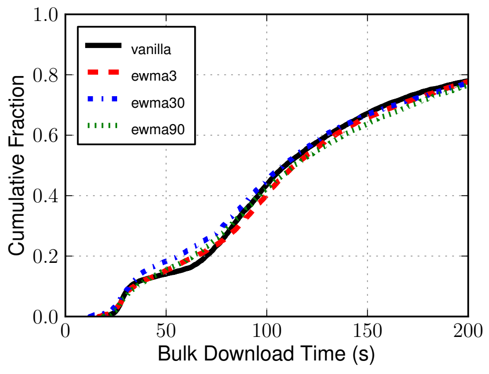
(c) 5 MiB, First Byte, Medium Load



(d) 5 MiB, Last Byte, Medium Load



(e) 5 MiB, First Byte, Heavy Load



(f) 5 MiB, Last Byte, Heavy Load

Figure 5.3: Performance comparison for the bulk clients (5 MiB) under "light," "medium," and "heavy" loads of 25, 50, and 100 bulk clients, respectively, in full-network deployments using the EWMA circuit scheduler and the vanilla Tor round-robin circuit scheduler. While the EWMA circuit scheduler works best under heavily loaded networks, there are half-life configurations that reduce client performance.

used under all network conditions since it is not clear that performance will always improve. When improvements over the round-robin scheduler are possible, they may be insignificant or depend on a correctly configured half-life. Tang and Goldberg find that low half-life values close to 0 and high values close to 100 result in little improvement when compared to unprioritized, vanilla Tor. We find this to be true under lighter loads, but Figure 5.2 shows that larger half-life values result in better performance for more heavily loaded networks. Our results illustrate that performance benefits are heavily dependent on network traffic patterns, and we stress the importance of frequently assessing the network to assist in determining appropriate half-life values over time. We suggest that more analysis is required to determine if the EWMA scheduler actually improves performance in the live Tor network, and if relays should enable it by default.

5.3 Circuit Scheduling with Proportional Differentiation

This section explores the differentiated services architecture (DiffServ) [86] as a way to improve control over the scheduling decisions made by Tor relays.

5.3.1 Proportional Differentiation Model

We aim to design algorithms that ensure relative quality metrics between qualifying and non-qualifying traffic using a priority scheduling mechanism based on the proportional differentiation model [34]. The model aims to provide *predictable* and *controllable* performance: quality metrics should be consistently proportional between classes and the proportions should be adjustable.

In the proportional differentiation model, traffic is separated into N classes labeled $c_1, \ldots, c_N$. The model states that the desired quality measurement q_i for each class c_i should be proportional to the other classes, where the proportions are configured with a differentiation parameter p_i for each class. The classes should be scheduled such that the relative quality of each class follow the configured differentiation parameters:

$$\forall i \in [N], \forall j \in [N] : \ \frac{q_i(t, t+\sigma)}{q_j(t, t+\sigma)} = \frac{p_i}{p_j} \tag{5.2}$$

where $p_1 < p_2 < \ldots < p_N$, p_i/p_j defines the desired quality proportion between class c_i and class c_j, and σ is the measurement timescale. Note that proportional quality differentiation is only defined under the condition that the model is feasible (i.e., when the amount and distribution of traffic allow a work-conserving scheduler to effect a difference). We will consider two schedulers that operate in this model.

Proportional Delay

Dovrolis *et al.* explore *Proportional Delay Differentiation* (PDD) and a priority scheduler that differentiates classes using queuing delay (packet waiting time) as the quality metric [87]. The scheduler utilizes two statistics to determine which class c_i to schedule at time t: the queuing delay $D_i(t)$ of the longest waiting packet in c_i, and the long-term average delay $\delta_i(t)$ of all previously scheduled packets (i.e., the average queuing delay of packets at the moment they are scheduled). The quality metric under Proportional Delay Differentiation becomes:

$$q_i'(t) = D_i(t) \cdot f + \delta_i(t) \cdot (1 - f) \tag{5.3}$$

where f is an adjustable fraction. A priority is computed for each c_i as $P_i'(t) = q_i'(t)/p_i(t)$, and the *longest* waiting packet from the class with the *maximum* computed priority is scheduled next. Once a class is selected, the delay differentiation approach is essentially first-come, first-served scheduling among the circuits belonging to that class since each packet's delay timer starts when the packet enters the queue.

Proportional Throughput

We also explore an alternative approach that may be better suited to scheduling in the Tor network. In particular, prioritizing circuits with a low exponentially-weighted moving average (EWMA) circuit throughput may improve performance of bursty traffic while minimally harming bulk traffic with higher desired long-term throughput [33]. Adhering to the model introduced by Dovrolis *et al.* , we explore *Proportional Throughput Differentiation* (PTD) and a scheduler that differentiates classes using the EWMA throughput as the quality metric. We define the quality metric at time t using the EWMA throughput of the lowest throughput circuit $T_i(t)$ and the long-term EWMA throughput $\tau_i(t)$ of previously scheduled circuits (i.e., the average of the average throughput of circuits at each moment they are scheduled):

$$q_i''(t) = T_i(t) \cdot f + \tau_i(t) \cdot (1 - f) \tag{5.4}$$

where f remains adjustable. The priority is computed for each c_i as $P_i''(t) = q_i''(t) \cdot p_i(t)$, and the circuit with the *lowest* EWMA throughput from the class with the *minimum* computed priority is scheduled next.

Assigning Classes

The DiffServ model outlined above requires that circuits are assigned to classes in order to differentiate performance based on quality metrics. Unfortunately, classifying traffic with deep packet inspection is not possible with encrypted Tor traffic, and requiring clients to specify their class may weaken anonymity. Instead, *statistical fingerprinting* techniques [90, 91, 92] may be used to classify traffic based solely on its statistical properties without input from other network nodes. Note that supervised classification techniques have been previously explored in the context of the Tor network [93], but unsupervised classification is still an open research problem.

In the evaluation of our schedulers, we will limit our focus to networks with "ideal" and "heuristic" classification techniques. In "ideal" classification, each Tor circuit is pre-labeled with its service class based on the type of traffic sent by the client. This information will improve our understanding of the *best possible performance* we can expect from our scheduling algorithms, and should only be worsened by less-accurate classifiers. In "heuristic" classification, we will classify circuits based on the total amount of data sent through them. Under the heuristic, all circuits start out assigned to the highest priority class c_1, and are reassigned to the next lower priority class c_{i+1} for each additional B bytes sent through the circuit. A circuit's priority will continue to be lowered until the circuit is assigned to the lowest priority class c_N. The proportional differentiation parameters p_i may be configured for each class as discussed above.

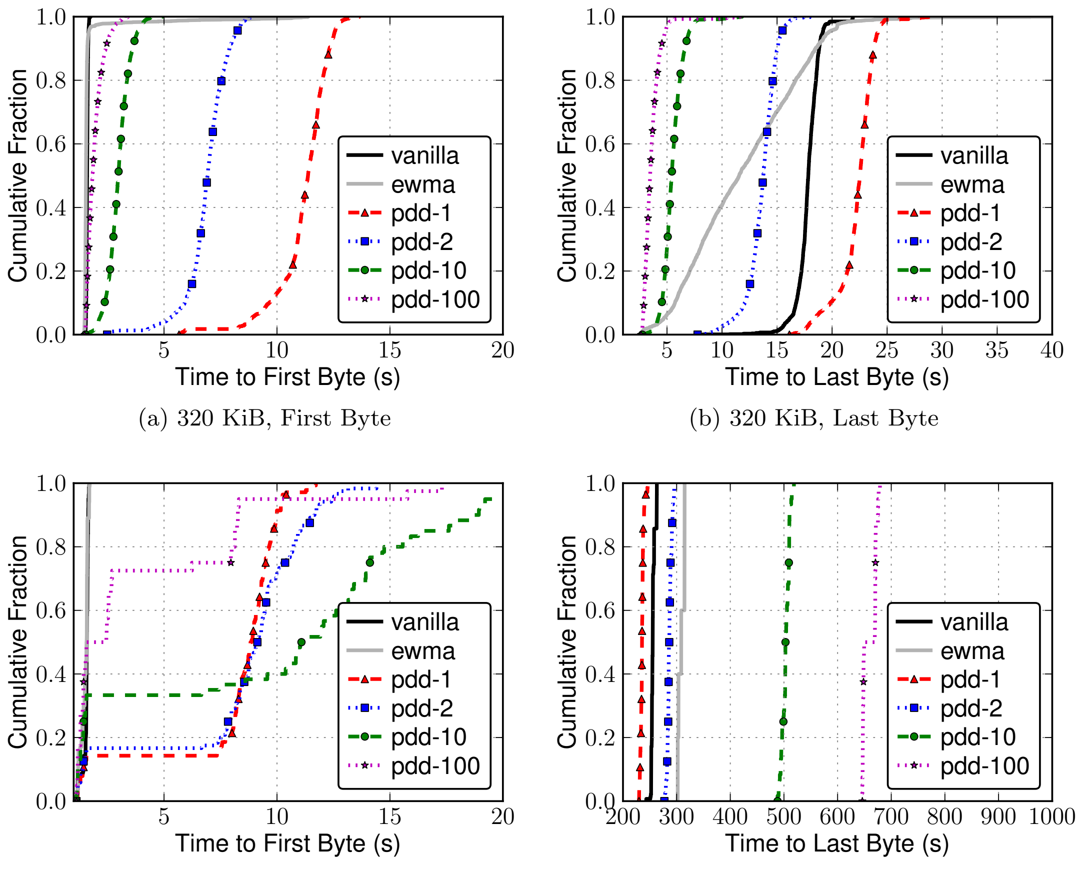
5.3.2 DiffServ in Single-Circuit Topology

Using Shadow, we now investigate and compare our proportional differentiation schedulers using "ideal" classification in a small single-circuit topology. Using a small topology enables us to isolate each scheduler's functionalities and better understand the fundamental improvements they may provide.

Our single-circuit network consists of 4 relays configured so that all the clients select the same relay as the circuit entry node, the same relay as the circuit exit node, and the same relay as the circuit middle node (the middle node also acts as the directory authority). The fourth node is only used to assist the relays in bootstrapping, but it is not selected for any client circuits. We also configure 20 web clients and 20 bulk clients. The web clients download 320 KiB files, pausing in between each download for a time chosen uniformly at random between 1 and 20 seconds. The bulk clients continuously download 5 MiB files without pausing. In each experiment, we configure all nodes to use the same scheduler: we test each of the original round-robin scheduler (vanilla), the EWMA scheduler of Tang and Goldberg (ewma), the proportional delay differentiation scheduler (pdd), and the proportional throughput differentiation scheduler (ptd). The clients download for 30 virtual minutes during the experiment, and we record the time to download the first and last byte of each file as performance metrics.

Figure 5.4 shows the results obtained with the proportional delay differentiation scheduler. The legend shows the various differentiation parameters (the desired ratios between the quality metrics of each class) that we tested. With no differentiation between classes (pdd-1), download times are worse than when scheduling with round-robin for web users (Figure 5.4(b)) but better for bulk users (Figure 5.4(d)), suggesting that bulk users may occasionally benefit from scheduling based on packet delays due to the larger number of cells in flight at any time. Increasing the differentiation increases priority for the web users in the first class (Figures 5.4(a) and 5.4(b)), but decreases it for the second class (Figure 5.4(d)). Our pdd results indicate that we can precisely specify the performance qualities we desire for each class, given that we satisfy the feasibility condition of the proportional differentiation model.

Figure 5.5 shows the results obtained with the proportional throughput differentiation scheduler. The legend again shows the various differentiation parameters that we



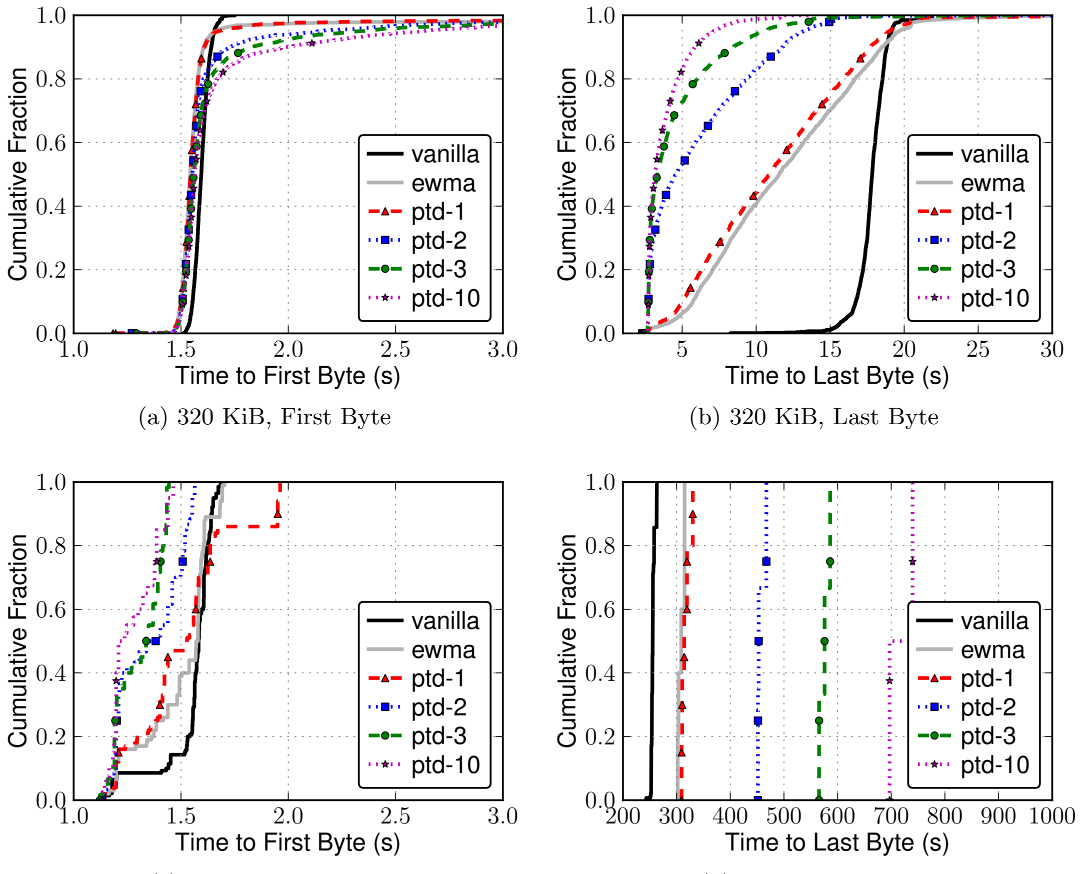
(a) 320 KiB, First Byte

(b) 320 KiB, Last Byte

(c) 5 MiB, First Byte

(d) 5 MiB, Last Byte

Figure 5.4: Proportional delay differentiation (pdd) performance comparison in our single-circuit network with 20 web clients and 20 bulk clients, using “ideal” classification, for various proportional differentiation parameters. While pdd without differentiation (pdd-1) prefers the bulk class slightly more than the round-robin scheduler, increasing the differentiation increases priority for the web clients in the first class (b) and decreases it for the bulk clients in the second class (d).



(a) 320 KiB, First Byte

(b) 320 KiB, Last Byte

(c) 5 MiB, First Byte

(d) 5 MiB, Last Byte

Figure 5.5: Proportional throughput differentiation (ptd) performance comparison in our single-circuit network with 20 web clients and 20 bulk clients, using "ideal" classification, for various proportional differentiation parameters. No differentiation reduces ptd to EWMA (ptd-1). Increasing the differentiation increases priority for the web clients in the first class (b) and decreases it for the bulk clients in the second class (d).

tested. With no differentiation between classes (ptd-1), the ptd scheduler essentially reduces to EWMA scheduling with a slight bias towards the circuits in the class with the lowest average throughput. This highlights the advantages of using EWMA throughput as the underlying quality metric: unlike pdd, the ptd scheduler still prefers bursty traffic to bulk traffic, even when there is no configured differentiation. As with pdd, increasing the differentiation increases priority for the web users in the first class (Figured 5.5(a) and 5.5(b)), but decreases it for the second class (Figure 5.5(d)). Our ptd results also indicate that we can precisely specify the performance qualities we desire for each class, given that we satisfy the feasibility condition of the proportional differentiation model.

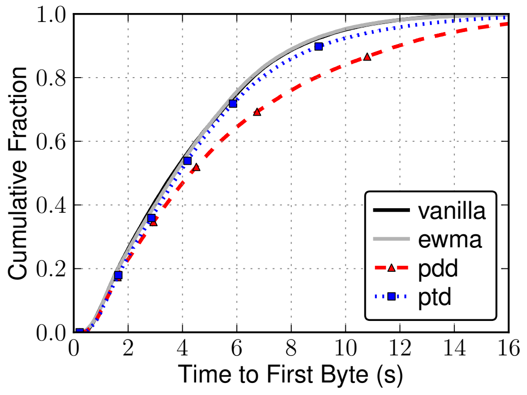
5.3.3 DiffServ in Full-Network Deployment

This section investigates our proportional differentiation schedulers using "heuristic" classification in a full-network topology. A full-network deployment will allow us to better understand the improvements that the scheduling algorithms may provide in practice under more realistic conditions.

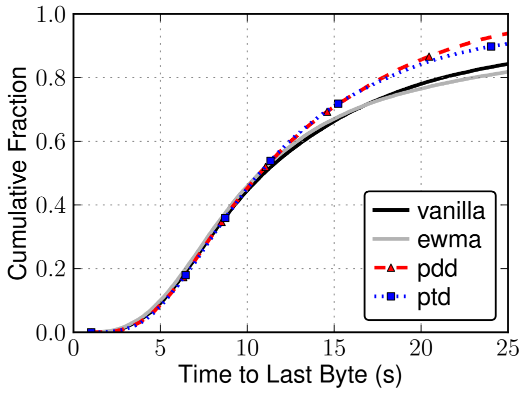
Our full-network setup consists of 100 relays (40 exit relays and 60 non-exit relays), 1000 clients (950 web clients and 50 bulk clients), and 100 servers. The behavior of the nodes is the same as specified above in Section 5.3.2. We also test a variety of client loads by changing the number of bulk nodes from 50 in the "normal load" configuration to 25 for a "light load" configuration, and 100 for a "heavy load" configuration.

Using our class assignment heuristic, we configure the pdd scheduler with 2 classes with a differentiation ratio of 1000, while each client is reassigned to the next lower priority class after sending 5 MiB through a circuit. We configure the ptd scheduler with 4 classes with a differentiation ratio of 10, while each client is reassigned to the next lower priority class after sending 1 MiB through a circuit. We choose these parameters because they tend to perform the best in our experiments, noting that future work should explore a deeper analysis of how the heuristics affect the scheduling mechanisms.

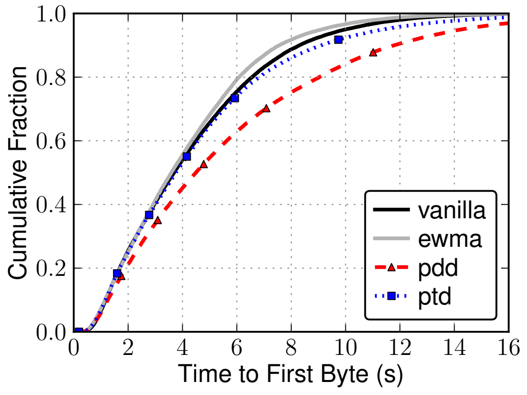
The results for "light", "medium", and "heavy" loads are shown in Figure 5.6 for web clients and Figure 5.7 for bulk clients. Figure 5.6 shows that as the load gets heavier (Figure 5.6(b) to Figure 5.6(d) to Figure 5.6(f)), the pdd scheduler tends to degrade overall web client performance, whereas the ptd scheduler tends to consistently outperform both the round-robin and ewma schedulers. This is expected: ptd is based



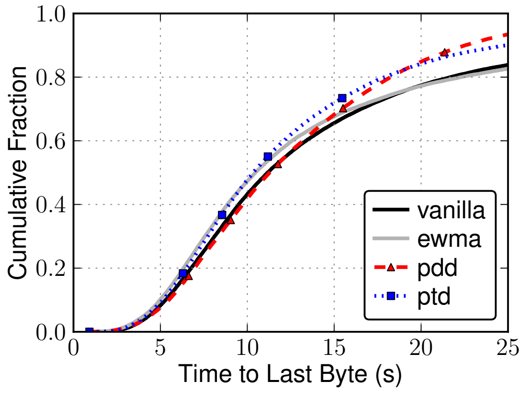
(a) 320 KiB, First Byte, Light Load



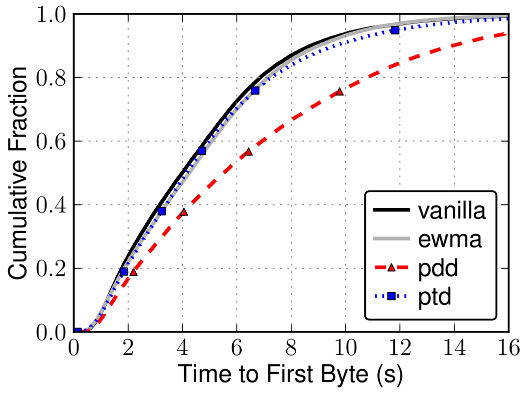
(b) 320 KiB, Last Byte, Light Load



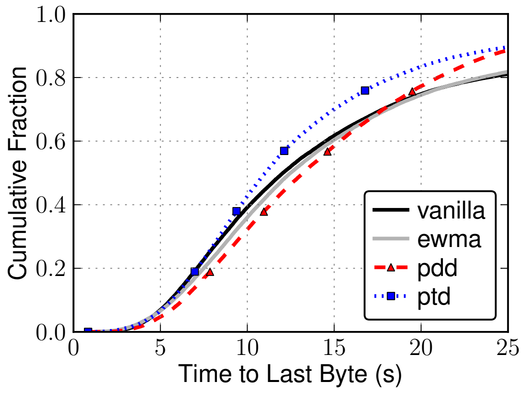
(c) 320 KiB, First Byte, Medium Load



(d) 320 KiB, Last Byte, Medium Load

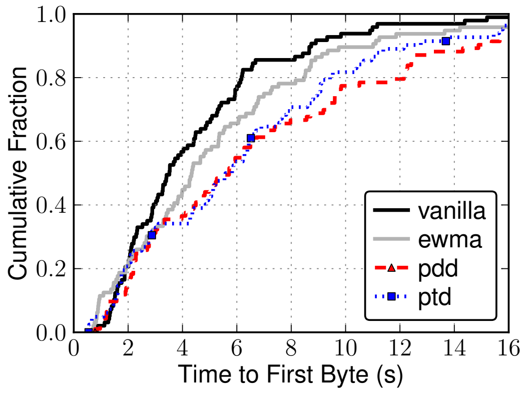


(e) 320 KiB, First Byte, Heavy Load

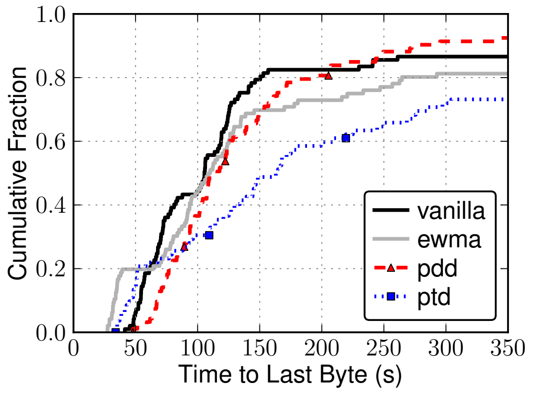


(f) 320 KiB, Last Byte, Heavy Load

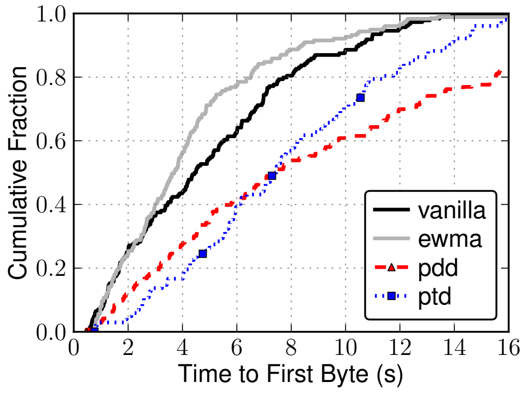
Figure 5.6: Performance comparison for the 950 web clients (320 KiB) under "light," "medium," and "heavy" loads of 25, 50, and 100 bulk clients, respectively, in full-network deployments using the DiffServ, EWMA, and vanilla round-robin circuit schedulers.



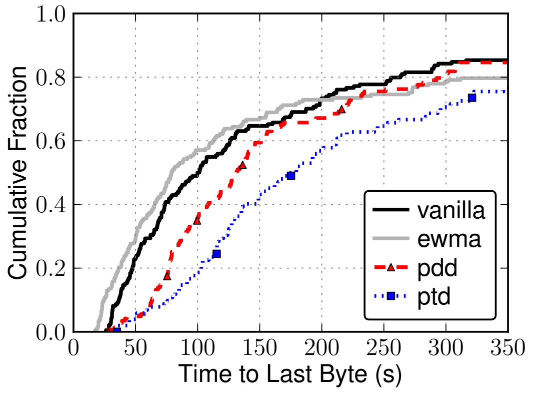
(a) 5 MiB, First Byte, Light Load



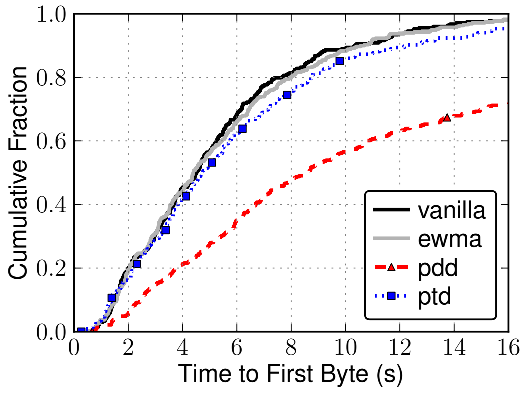
(b) 5 MiB, Last Byte, Light Load



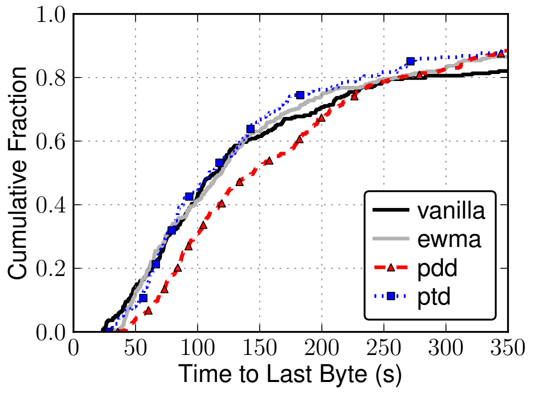
(c) 5 MiB, First Byte, Medium Load



(d) 5 MiB, Last Byte, Medium Load



(e) 5 MiB, First Byte, Heavy Load



(f) 5 MiB, Last Byte, Heavy Load

Figure 5.7: Performance comparison for the bulk clients (5 MiB) under “light,” “medium,” and “heavy” loads of 25, 50, and 100 bulk clients, respectively, in full-network deployments using the DiffServ, EWMA, and vanilla round-robin circuit schedulers.

on EWMA and we expect that web client performance should only increase by specifying a differentiation ratio that prioritizes them over bulk clients; pdd is based on queuing delays, and bulk clients that have clogged up a queue will have cells with higher delays that occasionally get prioritized over web client cells. Figure 5.7 shows that both pdd and ptd tend to achieve our goal of decreasing performance for bulk users. Although pdd reorders some bulk cells in front of web cells, it does not seem to affect the total download time for most bulk files.

5.4 Summary

As an example of the powerful capabilities of our simulation approach, we explored the EWMA circuit priority scheduler recently proposed and currently used in Tor to validate previous results and determine the effects of a network-wide deployment. We found that correct half-life configurations are highly network and load dependent, and that EWMA actually reduces performance for clients under certain network conditions. Although enabled by default, it is unclear if the scheduler improves performance in the live Tor network.

We then explored the proportional differentiation model as an alternative approach to Tor circuit scheduling. We evaluated schedulers based on proportional delay and proportional throughput in both single-circuit and full-network deployments, using ideal and heuristic classification techniques. We found that the proportional differentiation model provides mechanisms for precisely specifying the desired quality metrics for various classes of service, and showed the extent to which we can improve performance for bursty traffic.

Chapter 6

Throttling Tor Bandwidth Parasites

6.1 Introduction

Recall that Tor relays are run by volunteers located throughout the world and service hundreds of thousands of Tor clients [27] with high bandwidth demands. A relay's utility to Tor is dependent on both the bandwidth *capacity* of its host network and the bandwidth *restrictions* imposed by its operator (its possible to limit bandwidth contributions). Although bandwidth donations vary widely, the majority of relays offer less than 100 KiB/s and may become bottlenecks when chosen for a circuit. Bandwidth bottlenecks lead to network congestion and impair client performance.

Bottlenecks are further aggravated by bulk users, which make up roughly five percent of connections and forty percent of the bytes transferred through the network [28]. Bulk traffic increases network-wide congestion and punishes interactive users as they attempt to browse the web and run `SSH` sessions. Bulk traffic also constitutes a simple denial of service (DoS) attack on the network as a whole: with nothing but a moderate number of bulk clients, an adversary can intentionally significantly degrade the performance of the entire Tor network for most users. This is a malicious attack as opposed to an opportunistic use of resources without regard for the impact on legitimate users, and could be used by censors [31] to discourage use of Tor. Bulk traffic effectively averts potential users from Tor, decreasing both Tor's client diversity and anonymity [25, 32].

There are three general approaches to alleviate Tor's performance problems: increase network capacity; optimize resource utilization; and reduce network load.

6.1.1 Increasing Capacity

One approach to reducing bottlenecks and improving performance is to add additional bandwidth to the network from new relays. Previous work has explored recruiting new relays by offering performance incentives to those who contribute [46, 47, 94]. While these approaches show potential, they have not been deployed due to a lack of understanding of the anonymity and economic implications they would impose on Tor and its users. It is unclear how an incentive scheme will affect users' anonymity and motivation to contribute: Acquisti *et al.* [45] discuss how differentiating users by performance may reduce anonymity while competition may reduce the sense of community and convince users that contributions are no longer warranted.

New high-bandwidth relays may also be added by the Tor Project [12] or other organizations. While effective at improving network capacity, this approach is a short-term solution that does not scale. As Tor increases speed and bandwidth, it will likely attract more users. More significantly, it will attract more high-bandwidth and BitTorrent users, resulting in a *Tragedy of the Commons* [95] scenario: the bulk users attracted to the faster network will continue to leech the additional bandwidth.

6.1.2 Optimizing Resource Utilization

Another approach to improving performance is to better utilize the available network resources. Tor's path selection algorithm ignores the slowest small fraction of relays while selecting from the remaining relays in proportion to their available bandwidth. The path selection algorithm also ignores circuits with long build times [96], removing the worst of bottlenecks and improving usability. Congestion-aware path selection [97] is another approach that aims to balance load by using opportunistic and active client measurements while building paths. However, low bandwidth relays must still be chosen for circuits to mitigate anonymity problems, meaning there are still a large number of circuits with tight bandwidth bottlenecks.

Tang and Goldberg previously explored modifications to the Tor circuit scheduler in order to prioritize bursty (i.e. web) traffic over bulk traffic using an exponentially-weighted moving average (EWMA) of relayed cells [33]. Early experiments show small improvements at a single relay, but full-network experiments indicate that the new scheduler has an insignificant effect on performance (see Section 5.2). It is unclear how performance is affected when deployed to the live Tor network. This scheduling approach attempts to better utilize the available bandwidth for specific traffic classes, but does not reduce bottlenecks introduced by the massive amount of bulk traffic currently plaguing Tor.

6.1.3 Reducing Load

All of the previously discussed approaches attempt to increase performance, but none of them directly address performance degradation problems created by bulk traffic clients.

This chapter addresses these problems by adaptively throttling bulk data transfers at the client's entry into the Tor network.

We emphasize that throttling is *fundamentally different* than scheduling, and the distinction is important in the context of the Tor network. Schedulers optimize the utilization of available bandwidth by following policies set by the network engineer, allowing the enforcement of fairness among flows (e.g. max-min fairness [85, 98] or proportional fairness [99]). However, throttling may explicitly *under-utilize* local bandwidth resources by intentionally imposing restrictions on clients' throughput in order to reduce aggregate network load.

By reducing bulk client throughput in Tor, we effectively reduce the bulk data transfer rate through the network, resulting in *fewer bottlenecks* and a less congested, more responsive Tor network that can better handle the burstiness of web traffic. Tor has recently implemented token buckets, a classic traffic shaping mechanism [100], to statically (non-adaptively) throttle client-to-guard connections at a given rate [38], but currently deployed configurations of Tor do not enable throttling by default. Unfortunately, the throttling algorithm implemented in Tor requires static configuration of throttling parameters: the Tor network must determine network-wide settings that work well and update them as the network changes. Further, it is not possible to automatically tune each relay's throttling configuration with the current algorithm.

6.1.4 Contributions

To the best of our knowledge, we are the first to explore throttling algorithms that *adaptively adjust* to the fluctuations and dynamics of Tor and each relay independently without the need to adjust parameters as the network changes. We also perform the first detailed investigation of the performance and anonymity implications of throttling Tor clients.

In Section 6.2, we introduce and test three algorithms that dynamically and adaptively throttle Tor clients using a basic token bucket rate-limiter as the underlying throttling mechanism. Our new adaptive algorithms use local relay information to dynamically select which connections get throttled and to adjust the rate at which those connections are throttled. Adaptively tuned throttling mechanisms are paramount to our algorithm designs in order to avoid the need to re-evaluate parameter choices as network capacity or relay load changes. Our *bit-splitting* algorithm throttles each connection at an adaptively adjusted, but reserved and equal portion of a guard node's bandwidth, our *flagging* algorithm aggressively throttles connections that have historically exceeded the statistically fair throughput, and our *threshold* algorithm throttles connections above a throughput quantile at a rate represented by that quantile.

We implement our algorithms in Tor[1] and test their effectiveness at improving performance in large scale, full-network deployments. Section 6.3 compares our algorithms to static (non-adaptive) throttling under a varied range of network loads. We find that the effectiveness of static throttling is highly dependent on network load and configuration whereas our adaptive algorithms work well under various loads with no configuration changes or parameter maintenance: web client performance was improved for every parameter setting we tested. We conclude that throttling is an effective approach to achieve a more responsive network.

Having shown that our adaptive throttling algorithms provide significant performance benefits for web clients and have a profound impact on network responsiveness, Section 6.4 analyzes the security of our algorithms under adversarial attack. We discuss several realistic attacks on anonymity and compare the information leaked by each algorithm relative to unthrottled Tor. Against intuition, we find that throttling clients reduces information leakage and improves network anonymity while minimizing the false positive impact on honest users.

6.2 Throttling Client Connections

Client performance in Tor depends heavily on the traffic patterns of others in the system. A small number of clients performing bulk transfers in Tor are the source of a large fraction of total network traffic [28]. The overwhelming load these clients place on the network increases congestion and creates additional bottlenecks, causing interactive applications, such as instant messaging and remote `SSH` sessions, to lose responsiveness.

This section explores client throttling as a mechanism to prevent bulk clients from overwhelming the network. Although a relay may have enough bandwidth to handle all traffic locally, bulk clients that continue producing additional traffic cause bottlenecks

[1]Software patches for our algorithms have been made publicly available to the community [101].

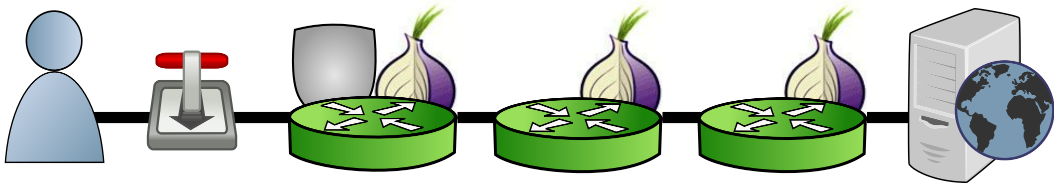


Figure 6.1: Throttling occurs at the connection between the client and guard to capture all streams to various destinations.

at other low-capacity relays. The faster a bulk downloader gets its data, the faster it will pull more into the network. Throttling bulk and other high-traffic clients prevents them from pushing or pulling too much data into the network too fast, reducing these bottlenecks and improving performance for the majority of users. Therefore, interactive applications and Tor in general will become much more usable, attracting new users who improve client diversity and anonymity.

We emphasize that throttling algorithms are not a replacement for congestion control or scheduling algorithms, although each approach may cooperate to achieve a common goal. Scheduling algorithms are used to *manage the utilization of bandwidth*, throttling algorithms *reduce the aggregate network load*, and congestion control algorithms attempt to do both. The distinction between congestion control and throttling algorithms is subtle but important: congestion control reduces *circuit load* while attempting to maximize network utilization, whereas throttling reduces *network load* in an attempt to improve circuit performance by explicitly under-utilizing connections to bulk clients using too many resources. Each approach may independently affect performance, and they may be combined to improve the network.

6.2.1 Static Throttling

Recently, Tor introduced the functionality to allow entry guards to throttle connections to clients [38] (see Figure 6.1). This client-to-guard connection is targeted because all client traffic (using this guard) will flow over this connection regardless of the number of streams or the destination associated with each.[2] The implementation uses a token bucket for each connection in addition to the global token bucket that already limits the total amount of bandwidth used by a relay. The size of the per-connection token buckets

[2]This work does not consider modified Tor clients.

can be specified using the `PerConnBWBurst` configuration option, and the bucket refill rate can be specified by configuring the `PerConnBWRate`. The configured throttling rate ensures that all client-to-guard connections are throttled to the specified long-term-average throughput while the configured burst allows deviations from the throttling rate to account for bursty traffic. The configuration options provide a static throttling mechanism: Tor will throttle all connections using these values until directed otherwise. Note that Tor does not enable or configure static throttling by default.

While static throttling is simple, it has two main drawbacks. First, static throttling requires constant monitoring and measurements of the Tor network to determine which configurations work well and which do not in order to be effective. We have found that there are many configurations of the algorithm that cause no change in performance, and worse, there are configurations that harm performance for interactive applications [102]. This is the opposite of what throttling is attempting to achieve. Second, it is not possible under the current algorithm to auto-tune the throttling parameters for each Tor relay. Configurations that appear to work well for the network as a whole might not necessarily be tuned for a given relay (we will show that this is indeed the case in Section 6.3). Each relay has very different capabilities and load patterns, and therefore may require different throttling configurations to be most useful.

6.2.2 Adaptive Throttling

Given the drawbacks of static throttling, we now explore and present three new algorithms that adaptively adjust throttling parameters according to local relay information. This section details our algorithms while Section 6.3 explores their effect on client performance and Section 6.4 analyzes throttling implications for anonymity.

There are two main issues to consider when designing a client throttling algorithm: *which connections* to throttle and at *what rate* to throttle them. The approach discussed above in Section 6.2.1 throttles *all* client connections at the *statically* specified rate. Each of our three algorithms below answers these questions *adaptively* by considering information local to each relay. Note that our algorithms dynamically adjust the `PerConnBWRate` while keeping a constant `PerConnBWBurst`.[3]

[3]Our experiments [102] indicate that a 2 MiB burst is ideal as it allows directory requests to be

Algorithm 3: Throttling clients by splitting bits.

```
 1: $B \leftarrow getRelayBandwidth()$
 2: $L \leftarrow getConnectionList()$
 3: $N \leftarrow L.length()$
 4: **if** $N > 0$ **then**
 5:    $splitRate \leftarrow \frac{B}{N}$
 6:    **for** $i \leftarrow 1$ to $N$ **do**
 7:       **if** $L[i].isClientConnection()$ **then**
 8:          $L[i].throttleRate \leftarrow splitRate$
 9:       **end if**
10:    **end for**
11: **end if**
```

Bit-splitting

A simple approach to adaptive throttling is to split a guard's bandwidth equally among all active client connections and throttle them all at this *fair split rate*. Therefore the `PerConnBWRate` will be adjusted as new connections are created or old connections are destroyed: more connections will result in lower rates. No connection will be able to use more than its allotted share of bandwidth unless it has unused tokens in its bucket. Inspired by Quality of Service (QoS) work from communication networks [103, 104, 105], bit-splitting will prevent bulk clients from unfairly consuming bandwidth and ensure that there is "reserved" bandwidth for web clients.

Note that Internet Service Providers employ similar techniques to throttle their customers, however, their client base is much less dynamic than the connections an entry guard handles. Therefore, our adaptive approach is more suitable to Tor. We include this algorithm in our analysis to determine what is possible with such a simple approach.

Flagging Unfair Clients

The bit-splitting algorithm focuses on adjusting the throttle rate and applying this to all client connections. Our next algorithm takes the opposite approach: configure a static throttling rate and adjust which connections get throttled. The intuition behind

downloaded unthrottled during bootstrapping while also throttling bulk traffic relatively quickly. The burst may need to be increased if the directory information grows beyond 2 MiB.

Algorithm 4: Throttling clients by flagging bulk connections, considering a moving average of throughput.

```
Require: $flagRate, \mathcal{P}, \mathcal{H}$
1:  $B \leftarrow getRelayBandwidth()$
2:  $L \leftarrow getConnectionList()$
3:  $N \leftarrow L.length()$
4:  $\mathcal{M} \leftarrow getMetaEWMA()$
5:  if $N > 0$ then
6:      $splitRate \leftarrow \frac{B}{N}$
7:      $\mathcal{M} \leftarrow \mathcal{M}.increment(\mathcal{H}, splitRate)$
8:      for $i \leftarrow 1$ to $N$ do
9:          if $L[i].isClientConnection()$ then
10:             if $L[i].EWMA > \mathcal{M}$ then
11:                 $L[i].flag \leftarrow True$
12:                 $L[i].throttleRate \leftarrow flagRate$
13:             else if $L[i].flag = True \wedge L[i].EWMA < \mathcal{P} \cdot \mathcal{M}$ then
14:                 $L[i].flag \leftarrow False$
15:                 $L[i].throttleRate \leftarrow infinity$
16:             end if
17:         end if
18:     end for
19: end if
```

this approach is that if we can properly identify the connections that use too much bandwidth, we can throttle them in order to maximize the benefit we gain per throttled connection. Therefore, our flagging algorithm attempts to classify and throttle bulk traffic while it avoids throttling web clients.

Since deep packet inspection is not desirable for privacy reasons, and is not possible on encrypted Tor traffic, we instead draw upon existing *statistical fingerprinting* classification techniques [90, 91, 92] that classify traffic solely on its statistical properties. When designing the flagging algorithm, we recognize that Tor already contains a statistical throughput measure for scheduling traffic on *circuits* using an exponentially-weighted moving average (EWMA) of recently sent cells [33]. We can use the same statistical measure on client *connections* to classify and throttle bulk traffic.

The flagging algorithm, shown in Algorithm 4, requires that each guard keeps an EWMA of the number of recently sent cells per client connection. The per-connection cell EWMA is computed in much the same way as the per-circuit cell EWMA: whenever the circuit's cell counter is incremented, so is the cell counter of the connection to which that circuit belongs. Note that clients can not affect others' per-connection EWMA since all of a client's circuits are multiplexed over a single throttled guard-to-client connection.[4] The per-connection EWMA is enabled and configured independently of its circuit counterpart.

We rely on the observation that bulk connections will have higher EWMA values than web connections since bulk clients are steadily transferring data while web clients "think" between each page download. Using this to our advantage, we can flag connections as containing bulk traffic as follows. Each relay keeps a single separate meta-EWMA $\mathcal{M}$ of cells transferred. $\mathcal{M}$ is adjusted by calculating the fair bandwidth split rate as in the bit-splitting algorithm, and tracking its EWMA over time. $\mathcal{M}$ does not correspond with any real traffic, but represents the upper bound of a connection-level EWMA if a connection were continuously sending only its fair share of traffic through the relay. Any connection whose EWMA exceeds $\mathcal{M}$ is flagged as containing bulk traffic and throttled.

There are three main parameters for the algorithm. As mentioned above, a per-connection half-life $\mathcal{H}$ allows configuration of the connection-level half-life independent of that used for circuit scheduling. $\mathcal{H}$ affects how long the algorithm remembers the amount of data a connection has transferred, and has precisely the same meaning as the circuit priority half-life [33]. Larger half-life values increase the ability to differentiate bulk from web connections while smaller half-life values make the algorithm more immediately reactive to throttling bulk connections. We would like to allow for a specification of the length of each penalty once a connection is flagged in order to recover and stop throttling connections that may have been incorrectly flagged. Therefore, we introduce a penalty fraction parameter $\mathcal{P}$ that affects how long each connection remains in a flagged and throttled state. If a connection's cell count EWMA falls below $\mathcal{P} \cdot \mathcal{M}$, its flag is removed and the connection is no longer throttled. Finally, the rate at which each flagged connection is throttled, i.e. the `FlagRate`, is statically defined and is not adjusted by the algorithm.

[4]The same is not true for the unthrottled connections between relays since each of them contain several circuits and each circuit may belong to a different client (see Chapter 2).

Algorithm 5: Throttling clients considering the loudest threshold of connections.

```
Require: $\mathcal{T}, \mathcal{R}, \mathcal{F}$
1: $L \leftarrow getClientConnectionList()$
2: $N \leftarrow L.length()$
3: if $N > 0$ then
4:     $selectIndex \leftarrow floor(\mathcal{T} \cdot N)$
5:     $L \leftarrow reverseSortEWMA(L)$
6:     $thresholdRate \leftarrow L[selectIndex].getMeanThroughput(\mathcal{R})$
7:     if $thresholdRate < \mathcal{F}$ then
8:         $thresholdRate \leftarrow \mathcal{F}$
9:     end if
10:    for $i \leftarrow 1$ to $N$ do
11:        if $i \leq selectIndex$ then
12:            $L[i].throttleRate \leftarrow thresholdRate$
13:        else
14:            $L[i].throttleRate \leftarrow infinity$
15:        end if
16:    end for
17: end if
```

Note that the flagging parameters need only be set based on system-wide policy and generally do not require independent relay tuning, but provides the flexibility to allow individual relay operators to deviate from system policy if they desire.

Throttling Using Thresholds

Recall the two main issues a throttling algorithm must address: selecting *which connections* to throttle and the *rate* at which to throttle them. Our bit-splitting algorithm explored adaptively adjusting the throttle rate and applying this to all connections while our flagging algorithm explored statically configuring a throttle rate and adaptively selecting the throttled connections. We now describe our final algorithm which attempts to adaptively address both issues.

The threshold algorithm also makes use of a connection-level cell EWMA, which is computed as described above for the flagging algorithm. However, EWMA is used here to sort connections by the loudest to quietest. We then select and throttle the loudest fraction $\mathcal{T}$ of connections, where $\mathcal{T}$ is a configurable threshold. For example, setting $\mathcal{T}$ to 0.1 means the loudest ten percent of client connections will be throttled. The selection is adaptive since the EWMA changes over time according to the amount of bandwidth consumed by each connection.

We have adaptively selected which connections to throttle and now must determine a throttle rate. To do this, we require that each connection tracks its throughput over time. We choose the average throughput rate of the connection with the minimum EWMA from the set of connections being throttled. For example, when $\mathcal{T} = 0.1$ and there are 100 client connections sorted from loudest to quietest, the chosen throttle rate is the average throughput of the tenth connection. Each of first ten connections is then throttled at this rate. In our prototype, we approximate the throughput rate as the average number of bytes transferred over the last $\mathcal{R}$ seconds, where $\mathcal{R}$ is configurable. $\mathcal{R}$ represents the time period between which the algorithm re-selects the throttled connections, adjusts the throttle rates, and resets each connection's throughput counters.

There is one caveat to the algorithm as described above. In our experiments in Section 6.3, we noticed that occasionally the throttle rate chosen by the threshold algorithm was zero. This would happen if the mean throughput of the threshold connection (line 6 in Algorithm 5) did not send data over the last $\mathcal{R}$ seconds. To prevent a throttle rate of zero, we added a parameter to statically configure a throttle rate floor $\mathcal{F}$ so that no connection is throttled below $\mathcal{F}$. Algorithm 5 details threshold adaptive throttling.

6.3 Experiments

In this section we explore the performance benefits possible with each throttling algorithm specified in Section 6.2. We perform experiments with Shadow [51, 52, 62], an accurate and efficient discrete event simulator that runs real Tor code over a simulated network. Shadow allows us to run an entire Tor network on a single machine and configure characteristics such as network latency, bandwidth, and topology. Since Shadow runs real Tor, it accurately characterizes application behavior and allows us to focus on experimental comparison of our algorithms. A direct comparison between Tor and Shadow-Tor performance is presented in Chapter 4 [62].

6.3.1 Experimental Setup

Using Shadow, we configure a private Tor network with 200 `HTTP` servers, 950 Tor web clients, 50 Tor bulk clients, and 50 Tor relays. The distribution of clients in our experiments approximates that found by McCoy *et al.* [28]. All of our nodes run inside the Shadow simulation environment.

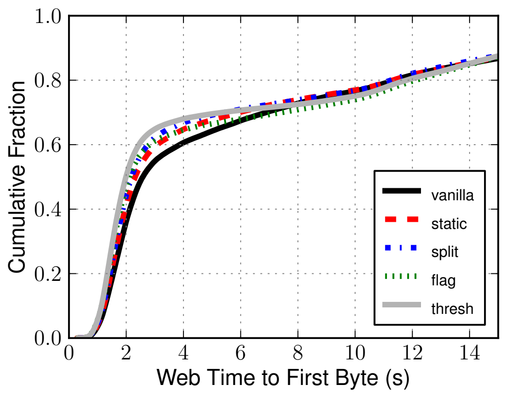
In our experiments, each client node runs Tor in client-only mode as well as an `HTTP` client application configured to download over Tor's `SOCKS` proxy available on the local interface. Each web client downloads a 320 KiB file[5] from a randomly selected one of our `HTTP` servers, and pauses for a length of time drawn from the UNC "think time" data set [58] before downloading the next file. Each bulk client repeatedly downloads a 5 MiB file from a randomly selected `HTTP` server without pausing. Clients track the time to the first and the last byte of the download as indications of network responsiveness and overall expected client performance.

Tor relays are configured with bandwidth parameters according to a Tor network consensus document.[6] We configure our network topology and latency between nodes according to the geographical distribution of relays and pairwise PlanetLab node ping times. Our simulated network mirrors a previously published Tor network model [62] that has been compared to and shown to closely approximate the load of the live Tor network [30].

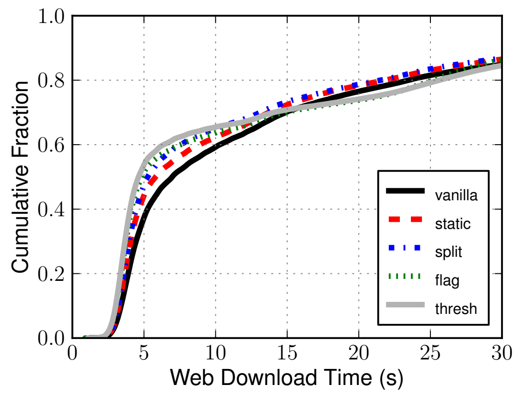
We focus on the time to the first data byte for web clients as a measure of network responsiveness, and the time to the last data byte—the download time—for both web and bulk clients as a measure of overall performance. In our results, "vanilla" represents unmodified Tor using a round-robin circuit scheduler and no throttling—the default settings in the Tor software—and can be used to compare relative performance between experiments. Each experiment uses network-wide deployments of each configuration. To further reduce random variances, we ran all configurations five times each. Therefore, every curve on every CDF shows the cumulative results of five experiments.

[5]The average webpage size reported by Google web metrics [80].

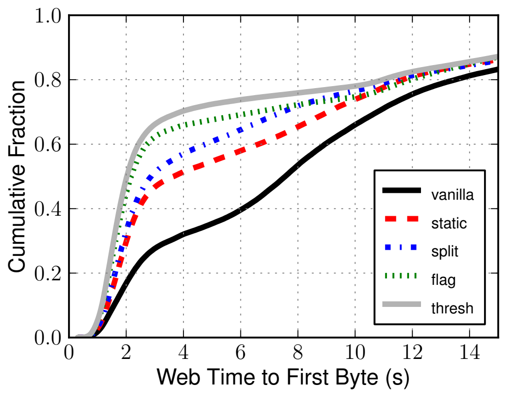
[6]Retrieved on 2011-04-27 and valid from 03-06:00:00



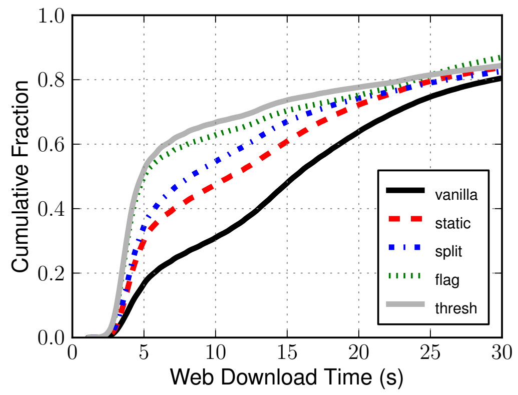
(a) 320 KiB, First Byte, Light Load



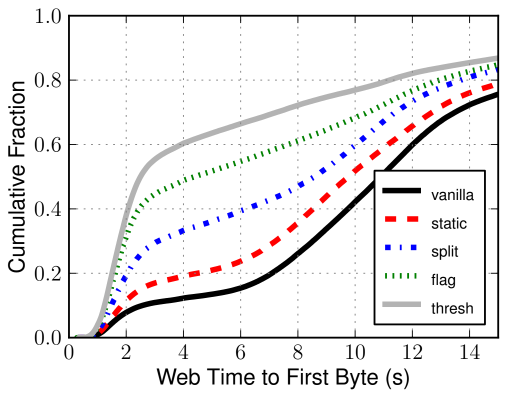
(b) 320 KiB, Last Byte, Light Load



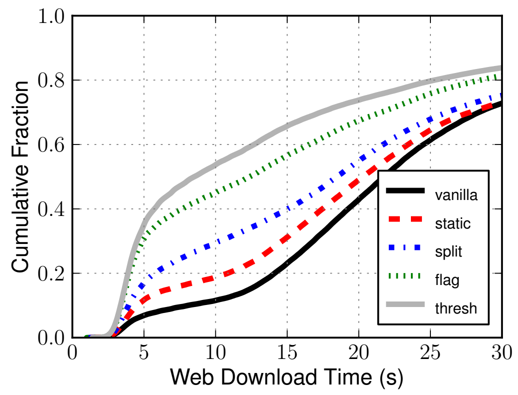
(c) 320 KiB, First Byte, Medium Load



(d) 320 KiB, Last Byte, Medium Load

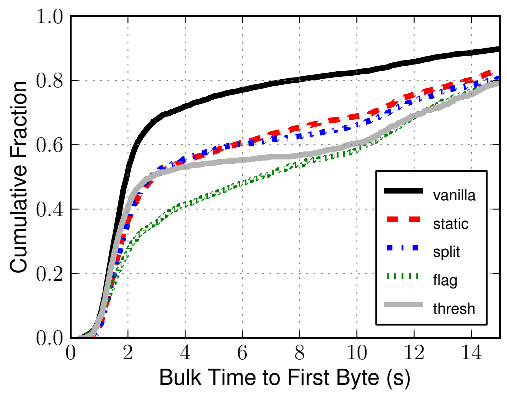


(e) 320 KiB, First Byte, Heavy Load

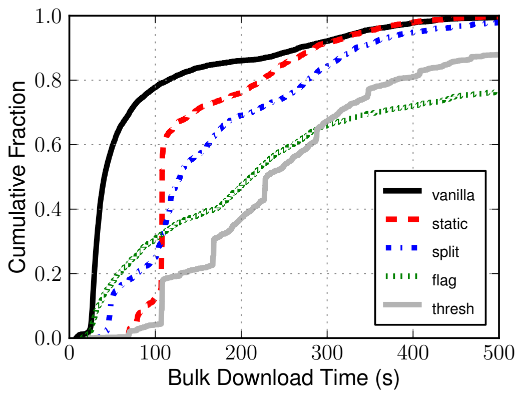


(f) 320 KiB, Last Byte, Heavy Load

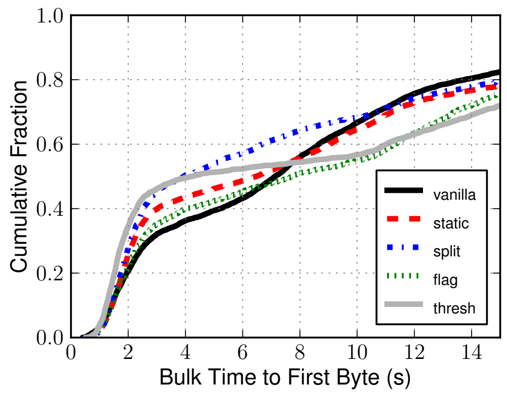
Figure 6.2: Performance comparison for the 950 web clients (320 KiB) under "light," "medium," and "heavy" loads of 25, 50, and 100 bulk clients, respectively.



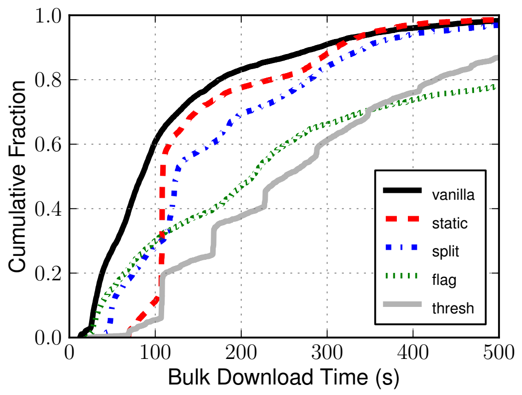
(a) 5 MiB, First Byte, Light Load



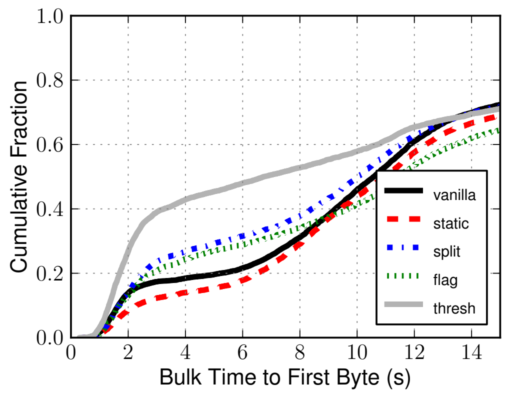
(b) 5 MiB, Last Byte, Light Load



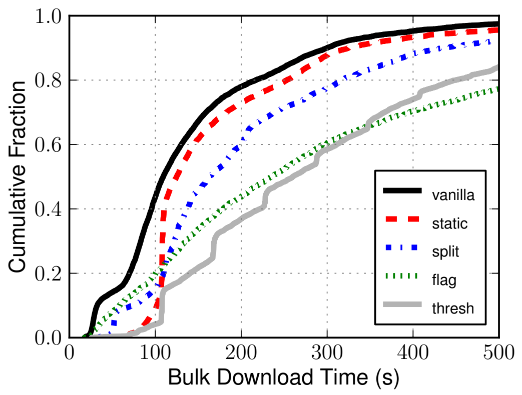
(c) 5 MiB, First Byte, Medium Load



(d) 5 MiB, Last Byte, Medium Load



(e) 5 MiB, First Byte, Heavy Load



(f) 5 MiB, Last Byte, Heavy Load

Figure 6.3: Performance comparison for the bulk clients (5 MiB) under "light," "medium," and "heavy" loads of 25, 50, and 100 bulk clients, respectively.

6.3.2 Results

Our results focus on the algorithmic configurations that we found to maximize web client performance [102] while we show how the algorithms perform when the network load varies from light (25 bulk clients) to medium (50 bulk clients) to heavy (100 bulk clients). The experimental setup is otherwise unmodified from the model described above. Running the algorithms under various networks with various loads allows us to highlight the unique and novel features each provides.

Figure 6.2 shows web client performance and Figure 6.3 shows bulk client performance for our algorithms. The time to first byte indicates network responsiveness for web clients while the download time indicates overall client performance for web and bulk clients. Client performance is shown for the lightly loaded, normally loaded, and heavily loaded networks. Overall, Figure 6.3 shows that static throttling results in the least amount of bulk traffic throttling while Figure 6.2 shows that it provides the lowest benefit to web clients. For the bit-splitting algorithm, Figure 6.2 shows improvements over static throttling for web clients for both time to first byte and overall download times, while Figure 6.3 shows that download times for bulk clients are also slightly increased. The flagging and the threshold throttling algorithms perform somewhat more aggressive throttling of bulk traffic and therefore also provide the greatest improvements in web client performance.

We find that each algorithm is effective at throttling bulk clients independent of network load, as evident in Figures 6.3(b), 6.3(d) and 6.3(f). However, performance benefits for web clients vary slightly as the network load changes. When the number of bulk clients is halved, throughput in Figure 6.2(b) is fairly similar across algorithms. However, when the number of bulk clients is doubled, responsiveness in Figure 6.2(e) and throughput in Figure 6.2(f) for both the static throttling and the adaptive bit-splitting algorithm lag behind the performance of the flagging and threshold algorithms. Static throttling would likely require a reconfiguration of throttling parameters while bit-splitting throttles less effectively than our flagging and threshold algorithms.

As seen in Figures 6.2(a), 6.2(c), and 6.2(e), as the load changes, the strengths of each algorithm become apparent. The flagging and threshold algorithms stand out as the best approaches for both web client responsiveness and throughput, and Figures 6.3(b), 6.3(d), and 6.3(f) show that they are also most aggressive at throttling bulk clients.

Table 6.1: Total data downloaded in our simulations by client type. Throttling reduces the bulk traffic share of the load on the network. The flagging algorithm is the best at throttling bulk traffic under light, medium, and heavy loads of 25, 50, and 100 bulk clients, respectively.

		vanilla	static	split	flag	thresh
light	Data (GiB)	88.3	80.3	78.3	72.1	69.8
	Web (%)	74.5	83.7	85.9	92.7	90.1
	Bulk (%)	25.5	16.3	14.1	7.3	9.9
medium	Data (GiB)	92.2	88.6	84.7	77.7	76.3
	Web (%)	65.8	72.4	75.0	86.2	82.8
	Bulk (%)	34.2	27.6	25.0	13.8	17.2
heavy	Data (GiB)	94.7	91.1	85.0	81.7	85.0
	Web (%)	55.8	60.5	64.3	75.4	71.2
	Bulk (%)	44.2	39.5	35.7	24.6	28.8

The flagging algorithm appears very effective at accurately classifying bulk connections regardless of network load. The threshold algorithm maximizes web client performance in our simulations among all loads and all algorithms tested, since it effectively throttles the worst bulk clients while utilizing extra bandwidth when possible. Both the threshold and flagging algorithms perform well over all network loads tested, and their usage in Tor would require little-to-no maintenance while providing significant performance improvements for web clients.

Aggregate download statistics are shown in Table 6.1. The results indicate that we are approximating the load distribution measured by McCoy *et al.* [28] reasonably well. The data also indicates that as the number of bulk clients in our simulation increases, so does the total amount of data downloaded and the bulk fraction of the total as expected. The data also shows that all throttling algorithms reduce the total network load. Static throttling reduces load the least, while our adaptive flagging algorithm is both the best at reducing both overall load and the bulk percentage of network traffic. Each of our adaptive algorithms are better at reducing load than static throttling, due to their ability to adapt to network dynamics. The relative difference between each algorithm's effectiveness at reducing load roughly corresponds to the relative difference in web client performance in our experiments, as we discussed above.

6.3.3 Discussion

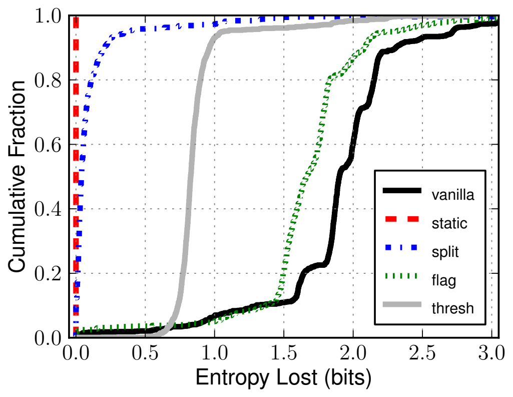
The best algorithm for Tor depends on multiple factors. Although not maximizing web client performance, bit-splitting is the simplest, the most efficient, and the most network neutral approach (every connection is allowed the same portion of a guard's capacity). This "subtle" or "delicate" approach to throttling may be favorable if supporting multiple client behaviors is desirable. Conversely, the flagging algorithm may be used to identify a specific class of traffic and throttle it aggressively, creating the potential for the largest increase in performance for unthrottled traffic. We are currently exploring improvements to our statistical classification techniques to reduce false positives and to improve the control over traffic of various types. For these reasons, we feel the bit-splitting and flagging algorithms will be the most useful in various situations. We suggest that perhaps bit-splitting is the most appropriate throttling algorithm to use initially, even if something more aggressive is desirable in the long term.

While requiring little maintenance, our algorithms were designed to use only local relay information. Therefore, they are incrementally deployable while relay operators may choose the desired throttling algorithm independent of others. Our algorithms are already implemented in Tor and software patches are available [101].

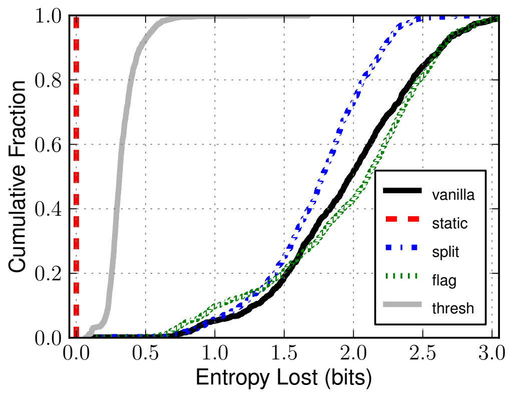
6.4 Analysis and Discussion

Having shown the performance benefits of throttling bulk clients in Section 6.3, we now analyze the security of throttling against adversarial attacks on anonymity. We will discuss the direct impact of throttling on anonymity: what an adversary can learn when guards throttle clients and how the information leaked affects the anonymity of the system. We lastly discuss potential strategies clients may use to elude the throttles.

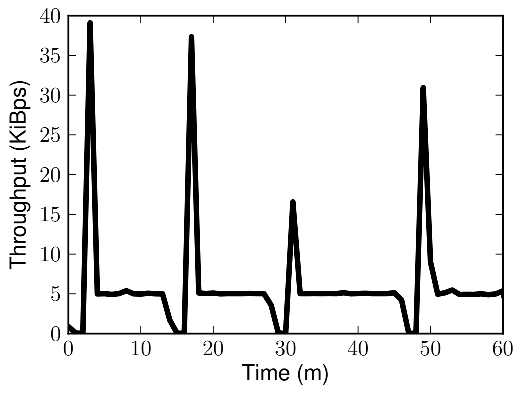
Before exploring practical attacks, we introduce two techniques an adversary may use to gather information about the network given that a throttling algorithm is enabled at all guards. Similar techniques used for throughput-based traffic analysis outside the context of throttling are discussed in detail by Mittal *et al.* [23]. Discussion about the security of our throttling algorithms in the context of practical attacks will follow.



(a) Leakage from Throughput Rate



(b) Leakage from Throttle Rate



(c) Probing Guards

Figure 6.4: Security analysis of our throttling algorithms. (a) Information leaked by learning circuit throughputs. (b) Information leaked by learning guards' throttle rates. (c) An adversary may discover the throttle rate by probing guards.

6.4.1 Gathering Information

Our analysis uses the following terminology. At time t, the throughput of a connection between a client and a guard is λ_t, the rate at which the client will be throttled is α_t, and the allowed data burst is β. Note that, as consistent with our algorithms, the throttle rate may vary over time but the burst is a static system-wide parameter.

Probing Guards

Using the above terminology, a connection is throttled if, over the last s seconds, its throughput exceeds the allowed initial burst and the long-term throttle rate:

$$\sum_{k=t-s}^{t} (\lambda_k) \geq \beta + \sum_{k=t-s}^{t} (\alpha_k) \tag{6.1}$$

A client may perform a simple technique to probe a specific guard node and determine the rate at which it gets throttled. The client may open a single circuit through the guard, selecting other high-bandwidth relays to ensure that the circuit does not contain a bottleneck. Then, it may download a large file and observe the change in throughput after receiving a burst of β payload bytes.

If the first β bytes are received at time t_1 and the download finishes at time $t_2 \geq t_1$, the throttle rate at any time t in this interval can be approximated by the mean throughput leading up to t:

$$\forall t \in [t_1, t_2],\ \alpha_t \approx \frac{\sum_{k=t_1}^{t} (\lambda_k)}{t - t_1} \tag{6.2}$$

Therefore, α_{t_2} approximates the actual throttle rate. Note that this approximation may under-estimate the actual throttle rate if the throughput falls below the throttle rate during the measured interval.

We simulate probing in Shadow [51, 52, 62] to show its effectiveness against the static throttling algorithm. As apparent in Figure 6.4(c), the throttle rate was configured at 5 KiB/s and the burst at 2 MiB. With enough resources, an adversary may probe every guard node to form a complete list of throttle rates.

Testing Circuit Throughput

A web server may determine the throughput of a connecting client's circuit by using a technique similar to that presented by Hopper *et al.* [19]. When the server gets an `HTTP` request from a client, it may inject either special JavaScript or a large amount of garbage `HTML` into a form element included in the response. The injected code will trigger a second client request after the original response is received. The server may adjust the amount of returned data and measure the time between when it sent the first response and received the second request to approximate the throughput of the circuit.

6.4.2 Adversarial Attacks

We now explore several adversarial attacks in the context of client throttling algorithms, and how an adversary may use those attacks to learn information and affect the anonymity of a client.

Attack 1

In our first attack, an adversary obtains a distribution on throttle rates by probing all Tor guard relays. We assume the adversary has resources to perform such an attack, e.g. by utilizing a botnet or other distributed network such as PlanetLab [48]. The adversary then obtains access to a web server and tests the throughput of a target circuit. With this information, the adversary may reduce the *anonymity set* of the circuit's potential guards by eliminating those whose throttle rate is inconsistent with the measured throughput.

This attack is somewhat successful against all of the throttling algorithms we have described. For bit-splitting, the anonymity set of possible guard nodes will consist of those whose bandwidth and number of active connections would throttle to the throughput of the target circuit or higher. By running the attack repeatedly over time, an intersection will narrow the set to those whose throttle rate is consistent with the target circuit throughput at all measured times.

The flagging algorithm throttles all flagged connections to the same rate system-wide. (We assume here that the set of possible guards is already narrowed to those whose bandwidth is consistent with the target circuit's throughput irrespective of throttling.) A circuit whose throughput matches the system-wide rate is either flagged at some guard or just coincidentally matches the system-wide rate and is not flagged because its EWMA has remained below the `splitRate` (see Algorithm 4) for its guard long enough to not be flagged or become unflagged. The throttling rate is thus not nearly as informative as for bit-splitting. If we run the attack repeatedly however, we can eliminate from the anonymity set any guard such that the EWMA of the target circuit should have resulted in a throttling but did not. Also, if the EWMA drops to the throttling rate at precise times (ignoring unusual coincidence), we can eliminate any guard that would not have throttled at precisely those times. Note that this determination must be made after the fact to account for the burst bucket of the target circuit, but it can still be made precisely.

The potential for information going to the attacker in the threshold algorithm is a combination of the potential in each of the above two algorithms. The timing of when a circuit gets throttled (or does not when it should have been) can narrow the anonymity set of entry guards as in the flagging algorithm. Once the circuit has been throttled, then any fluctuation in the throttling rate that separates out the guard nodes can be used to further narrow the set. Note that if a circuit consistently falls below the throttling rate of all guards, an attacker can learn nothing about its possible entry guard from this attack. Attack 2 considerably improves the situation for the adversary.

We simulated this attack in Shadow. An adversary probes all guards and forms a distribution on the throttle rate at which a connection would become throttled. We then form a distribution on circuit throughputs over each minute, and remove any guard whose throttle rate is outside a range of one standard deviation of those throughputs. Since there are 50 guards, the maximum entropy is $\log_2(50) \approx 5.64$; the entropy lost by this attack for various throttling algorithms relative to vanilla Tor is shown in Figure 6.4(a). We can see that the static algorithm actually loses no information, since all connections are throttled to the same rate, while vanilla Tor without throttling actually loses *more* information than any of the throttling algorithms. Therefore, the distribution on guard bandwidth leaks more information than throttled circuits' throughputs.

Attack 2

As in Attack 1, the adversary again obtains a distribution on throttle rates of all guards in the system. However, the adversary slightly modifies its circuit testing by continuously sending garbage responses. The adversary adjusts the size of each response so that it may compute the throughput of the circuit over time and approximates the rate at which the circuit is throttled. By comparing the estimated throttle rate to the distribution on guard throttle rates, the adversary may again reduce the anonymity set by removing guards whose measured throttle rate is inconsistent with the estimated rate.

For bit-splitting, by raising and lowering the rate of garbage sent, the attacker can match this with the throttled throughput of each guard. The only guards in the anonymity set would be those that share the same throttling rate that matches the flooded circuit's throughput at all times. To maximize what he can learn from flagging, the adversary should raise the EWMA of the target circuit at a rate that will allow him to maximally differentiate guards with respect to when they would begin to throttle a circuit. If this does not uniquely identify the guard, he can also use the rate at which he diminishes garbage traffic to try to learn more from when the target circuit stops being throttled. As in Attack 1 from the threshold algorithm, the adversary can match the signature of both fluctuations in throttling rate over time and the timing of when throttling is applied to narrow the set of possible guards for a target circuit.

We simulated this attack using the same data set as Attack 1. Figure 6.4(b) shows that a connection's throttle rate generally leaks slightly more information than its throughput. As in Attack 1, guards' bandwidth in our simulation leaks more information than the throttle rate of each connection for all but the flagging algorithm.

Attack 3

An adversary controlling two malicious servers can link streams of a client connecting to each of them at the same time. The adversary uses the circuit testing technique to send a response of $\frac{\beta}{2}$ bytes in size to each of two requests. Then, small "test" responses are returned after receiving the clients' second requests. If the throughput of each circuit when downloading the "test" response is consistently throttled, then it is possible that the requests are coming from the same client. This attack relies on the observation that all traffic on the same client-to-guard connection will be throttled at the same time since each connection has a single burst bucket.

This attack is intended to indicate if and when a circuit is throttled, rather than the throttling rate. It will therefore not be effective against bit splitting, but will work against flagging or threshold throttling.

Attack 4

Our final attack is an active denial of service attack that can be used to confirm a circuit's entry guard with high probability. In this attack, the adversary attempts to adjust the throttle rate of each guard in order to identify whether it carries a target circuit. An adversary in control of a malicious server may monitor the throughput of a target circuit over time, and may then open a large number of connections to each guard node until a decrease in the target circuit's throughput is observed. To confirm that a guard is on the target circuit, the adversary can alternate between opening and closing guard connections and continue to observe the throughput of the target circuit. If the throughput is consistent with the adversary's behavior, it has found the circuit's guard with high probability.

The one thing not controlled by the adversary in Attack 2 is a guard's criterion for throttling at a given time – `splitRate` for bit splitting and flagging and `selectIndex` for threshold throttling (see Algorithms 3, 4, and 5). All of these are controlled by the number of circuits at the guard, which Attack 4 places under the control of the adversary. Thus, under Attack 4, the adversary will have precise control over which circuits get throttled at which rate at all times and can therefore uniquely determine the entry guard.

Note that all of Attacks 1, 2, and 4 are intended to learn about the possible entry guards for an attacked circuit. Even if completely successful, this does not fully de-anonymize the circuit. But since guards themselves are chosen for persistent use by a client, they can add to pseudonymous profiling and can be combined with other information, such as that uncovered by Attack 3, to either reduce anonymity of the client or build a richer pseudonymous profile of it.

6.4.3 Eluding Throttles

A client may try multiple strategies to avoid being throttled. A client may instrument its downloading application and the Tor software to send application data over multiple Tor circuits. However, these circuits will still be subject to throttling since each of them uses the same throttled TCP connection to the guard. A client may avoid this by attempting to create multiple TCP connections to the guard. In this case, the guard may easily recognize that the connection requests come from the same client and can either deny the establishment of multiple connections or aggregate the accounting of all connections to that client. A client may use multiple guard nodes and send application data over each separate guard connection, but the client significantly decreases its anonymity by subverting the guard mechanism [15, 13]. Finally, the client could run and use its own guard node and avoid throttling itself. Although this strategy may actually benefit the network since it reduces the amount of Tor's capacity consumed by the client, the cost of running a guard may be sufficient to prevent its wide-scale adoption (see Chapter 7 [46] and Chapter 8 [106] for a discussion of incentives for running Tor relays).

Its important to note that the "cheating" techniques outlined above do not decrease the *security* or *performance* below what unthrottled Tor provides. At worst, even if all clients somehow manage to elude the throttles, performance and security both regress to that of unthrottled Tor. In other words, throttling can only *improve* the situation whether or not "cheating" occurs in practice.

6.5 Summary

This chapter analyzed client throttling by guard relays to reduce Tor network bottlenecks and improve responsiveness. We explored static throttling configurations while designing, implementing, and evaluating three new throttling algorithms that adaptively select which connections get throttled and dynamically adjust the throttle rate of each connection. Our adaptive throttling techniques use only local relay information and are considerably more effective than static throttling since they do not require re-evaluation of throttling parameters as network load changes. We found that client throttling is effective at both improving performance for interactive clients and increasing Tor's network resilience. We also analyzed the effects throttling has on anonymity and discussed the security of our algorithms against realistic adversarial attacks. We found that throttling improves anonymity: a guard's bandwidth leaks more information about its circuits when throttling is *disabled*.

Chapter 7

Recruiting New Tor Relays with BRAIDS

7.1 Introduction

Recall that the aggregate bandwidth costs of sending communication securely through multiple Tor relays are significantly higher than direct communication: the amount of bandwidth expended by a client is also expended by each relay in its circuit. A significant characteristic of communication over Tor is that *most clients use Tor for interactive applications* like web browsing, but *most data is transferred for non-interactive applications* like file sharing [28]. Moreover, Tor relays forward traffic for multiple circuits simultaneously, further increasing bandwidth obligations. The combination results in overloaded relays and drastically increased latency for communication over Tor [28].

A lack of incentives to run relays combined with the associated costs has hindered relay enlistment, and in turn, Tor's scalability. Although relaying traffic can increase user anonymity by frustrating attempts to differentiate relay-sourced from relay-forwarded data, there are no measurable benefits to providing service for others. Consequently, clients greatly outnumber relays in Tor. In 2009, there were an estimated 100,000 simultaneously active Tor clients [27] but only about 1,500 Tor relays.[1] This uneven distribution of bandwidth responsibilities combined with the disproportionately high client-to-relay ratio results in poor system performance and a *tragedy of the commons* [95] scenario: as Tor grows, it will require additional relays to provide bandwidth and traffic forwarding services to remain usable.

7.1.1 Recruiting New Relays

A significant problem faced by the current Tor system is *how to recruit new relays* to support expansion and ease the load suffered by current relays. There have been few approaches to solve the relay recruiting problem. One approach is to simply require every client to also be a relay, effectively reducing the client-to-relay ratio to 1:1 [107]. While we wish to promote relaying traffic, we do not wish to forcefully impose it: clients who are unable to run a relay due to censorship [108] would not be able to effectively use the system. Further, clients with poor Internet connectivity or a slow connection may be practically unable to provide service and may even harm network performance for others. Denying anonymity to these clients not only opposes the "anonymity for

[1]The corresponding client-to-relay ratio at the time this research was conducted (2009) was 66:1.

all" ideology, but also decreases anonymity for others since it reduces the diversity and size of the anonymity set [45] of potential circuit initiators. Tor's approach thus far has been to build a community and educate users about the benefits of anonymity, while simplifying relay setup and maintenance procedures. While this approach has been effective at expanding the network to its current size, relays are still in high demand and performance remains poor.

7.1.2 Introducing BRAIDS

In this chapter we present BRAIDS,[2] a set of practical mechanisms for the Tor anonymity network. BRAIDS increases incentives for relays while limiting the delays caused by non-interactive BitTorrent clients and keeping the system usable for everyone. Relays using BRAIDS enjoy lower latency and higher throughput than other users. In particular, BRAIDS allows relays to achieve 75% lower latency than non-relays for interactive web traffic – a 40% improvement over the current Tor network. Relays initiating non-interactive traffic receive a 90% increase in total bandwidth utilization from non-relays.

To improve performance, BRAIDS incorporates differentiated services and a scheduler based on the proportional differentiation model introduced by Dovrolis *et al.* [34, 109, 110, 87]. BRAIDS aggregates traffic into three hierarchical service classes *proportionally prioritized* as low-latency > high-throughput > normal, where the "cost" of high-throughput > low-latency (normal service is free). Each relay rate-limits the low-latency class to prevent high-throughput nodes from overwhelming low-latency traffic. Finally, traffic is paid and proportionally prioritized in *both* directions through the circuit, capturing Tor's asymmetric bandwidth requirements.

BRAIDS users optionally and anonymously "pay" relays with generic tickets that are both distributed freely in small amounts to all clients and relays, and collected by each relay while volunteering bandwidth to Tor. We use *relay-specific tickets* [111, 112] – random numbers combined with relay-identifiers – that are signed by an authority. Signed tickets are verified at the relay, defeating the double spending problem in which clients must make immediate deposits to catch cheaters that duplicate and spend a ticket multiple times. Information leakage is avoided since relays can verify tickets without assistance from an external entity. Tickets are valid during uniform intervals to prevent

[2]BRAIDS stands for "**B**andwidth **R**eciprocity **A**nd **I**ncentivized **D**ifferentiated **S**ervices."

linking clients with tickets. Clients who cannot or choose not to pay receive slightly reduced performance.

Other incentive-based recruitment approaches exist in the literature: the gold star scheme [47] gives preferential treatment to fast relays whereas PAR [113] and XPay [114] use e-cash and an online bank to produce monetary incentives. A variety of attacks [17, 19, 115, 20] make it difficult to design a secure solution with minimal loss of anonymity. In particular, bandwidth accounting mechanisms that give better service to relays that volunteer more bandwidth [47] in some cases significantly decrease the anonymity set of relays receiving better service, and in others [113] unintentionally allow an adversary to link relays to the same circuit.

BRAIDS is secure, retaining all of Tor's anonymity for users browsing the web, whereas the previously proposed gold star scheme [47] achieves less than 65%. Our anonymous ticket approach mitigates the intersection attack that has plagued previous schemes. Further, BRAIDS bounds cheating in such a way that users running relays must volunteer a significant amount of bandwidth before maliciously gaining a relatively insignificant number of tickets.

7.1.3 Outline

The remainder of the chapter is outlined as follows. In Section 7.2, we briefly discuss BRAIDS system requirements while detailing the design in Section 7.3. Analysis of security and parameters is given in Section 7.4, while simulations and results are described in Section 7.5. Finally, Section 7.6 concludes.

7.2 Requirements

BRAIDS' main goal is to encourage Tor clients to run relays by providing incentives in the form of increased performance. This system should prioritize low-latency traffic over high-throughput traffic to reduce the negative impact that file sharing users have on overall system performance while remaining usable by everyone. The service received by web browsing clients should not reduce their anonymity.

BRAIDS shares the same threat model as Tor – a local adversary who cannot observe or interfere with traffic sent between honest nodes. While we do not defend against current attacks on Tor, our system should not reduce Tor's security by introducing any new vulnerabilities. We should not leak information about the circuit initiator or the identities of relays composing the circuit.

In addition to the aforementioned entities, we introduce a centralized, partially-trusted, offline bank to manage and certify bandwidth accounting tasks. The bank should only be trusted to follow protocol, but we assume it can otherwise attack the system using any information in its possession. BRAIDS should provide accounting mechanisms for both the outgoing path from client to server, and the reverse path from server to client (previous systems [113, 114] do not provide payment mechanisms for the reverse path of a two-way communication channel), since many existing applications (e.g. web browsing and streaming media) have significantly higher downstream than upstream client requirements. Bandwidth accounting should be anonymous to protect the client's identity, while payments must be unforgeable, non-reusable, and should not be linkable to the client [116, 117]. Additionally, we require double spending prevention in the form of immediate double spending detection. Clients attempting to double-spend should not receive service. Any attempts to cheat the system should be bounded so that the overall efforts required to cheat will outweigh the achievable benefits.

Finally, our system should be an incrementally deployable extension to Tor: users transitioning from legacy software should not be partitioned from the network.

7.3 System Design

BRAIDS motivates users to operate Tor relays by introducing generic tickets for service accounting. Using blind signatures, users remain anonymous while obtaining a limited amount of free tickets from the bank. Tickets are then embedded into Tor cells to request the desired class of service – either low-latency and low-throughput (e.g. general web browsing) or high-latency and high-throughput (e.g. downloading or sharing large files). Each relay verifies its tickets to prevent double spending.

7.3.1 Relay-specific Tickets

Our ticket design draws upon ideas from coin ripping [111] and fair exchange for mix-nets [112]. Since tickets are relay-specific, our construction requires that clients have *a* *priori* knowledge about their desired communication partners [118]. Tor already requires knowledge of relays when building circuits, so relay-specific tickets are a natural choice.

Ticket Structure

A ticket T consists of a main part T_s, called the ticket *stub*, and a receipt part T_r, called the ticket *receipt*. The ticket stub contains the identity of the relay $\mathcal{R}$ (its public key) to which the ticket may be transferred. Letting | denote concatenation, we define a ticket for $\mathcal{R}$ as:

$$T^{\mathcal{R}} = \{T_s^{\mathcal{R}} \mid T_r^{\mathcal{R}}\} = \{\mathcal{R} \mid H(T_r^{\mathcal{R}}) \mid d \mid \sigma \mid T_r^{\mathcal{R}}\} \tag{7.1}$$

where H is a cryptographically secure one-way hash function, d is a set of date-stamps, σ is the bank's partially blind signature on $\{\mathcal{R} \mid H(T_r^{\mathcal{R}}) \mid d\}$, and $T_r^{\mathcal{R}}$ is a random bit-string used as a receipt.

Ticket Activation

We use a blind signature scheme [119] to activate tickets and ensure no information about relay $\mathcal{R}$ chosen by client $\mathcal{C}$ is revealed. Specifically, our construction uses a partially blind signature [120] where the client blinds information about the chosen relay $\mathcal{R}$. The bank attaches uniform public date-stamps (described below) to the ticket, but cannot discover the blinded relay information. The bank's signature creates a strongly unforgeable ticket $T^{\mathcal{R}}$, i.e. modifying the signed contents invalidates the ticket.

Ticket Validity Intervals

The bank attaches a set of date-stamps $d = \{d_u \mid d_v \mid d_w\}$ to the blinded relay information before signing. The time from ticket generation until the first date-stamp specifies the *spending interval* [-,d_u) in which relay-bound tickets may be spent. The time between the first and second date-stamp specifies the *relay-exchange interval* [d_u,d_v) in which a relay may exchange tickets at the bank for new relay-bound tickets. The time between the second and last date-stamp specifies the *client-exchange interval* [d_v,d_w) in which any client or relay may exchange tickets at the bank for new relay-bound tickets. Finally, tickets expire and are completely void after the final date-stamp. We suggest values for these parameters in Section 7.4.1.

The relay-before-client exchange priority prevents a client from maliciously exchanging a spent ticket before the relay can (causing the relay's ticket to appear double-spent upon attempted exchange) while still allowing the client to exchange unspent tickets. The final date-stamp prevents the bank's ticket database from growing infinitely large. The bank's global date-stamps are used for every ticket signed during a given time period to prevent the bank from marking tickets and linking clients with relays.

7.3.2 Ticket Transferability

Users may wish to transfer tickets to other users, or update tickets that have passed their spending interval but are not yet void. Unspent tickets may be transferred to relays for payment, but spent tickets or those past their spending interval must be first exchanged at the bank.

Users remain anonymous by exchanging tickets with the bank through Tor. However, in order to exchange during the relay-exchange interval, a relay is required to prove knowledge of its private key to the bank. Although this means relays are not anonymous in the exchange, we note that the bank can already enumerate the list of relays by downloading the public directory. The bank validates that the relay is bound to the exchanged tickets.

Relay Ticket Exchange

When relay $\mathcal{C}$ receives ticket $T^{\mathcal{C}}$, it becomes a *voucher* for $\mathcal{C}$ redeemable for a new relay-specific ticket. Relay $\mathcal{C}$ and bank $\mathcal{B}$ use Algorithm 6, `Relay-Ticket-Exchange`, to generate a new ticket spendable at relay $\mathcal{R}$ given that $\mathcal{C}$ presents a valid ticket voucher $T^{\mathcal{C}}$ and new ticket material. $\mathcal{C}$ performs setup on lines 1–2 by generating a random value and its hash. $\mathcal{C}$ sends $\mathcal{B}$ the voucher $T^{\mathcal{C}}$, and $\mathcal{B}$ is responsible for validating $T^{\mathcal{C}}$. $\mathcal{B}$ does this by verifying $T^{\mathcal{C}}$ is within the allowable date interval for relay-exchange, the identity $\mathcal{C}$ from $T^{\mathcal{C}}$ matches the real identity $\mathcal{C}$, $\mathcal{C} \mid H(T_r^{\mathcal{C}})$ never appeared before (i.e. $T^{\mathcal{C}}$ was not double spent), σ is a valid signature on $\mathcal{C} \mid H(T_r^{\mathcal{C}})$, and that a freshly computed $H(T_r^{\mathcal{C}})$ matches the hash from $T^{\mathcal{C}}$. If the voucher validates, $\mathcal{B}$ and $\mathcal{C}$ cooperate to produce a partially blind signature on the new ticket $T^{\mathcal{R}}$ payable to relay $\mathcal{R}$.

Algorithm 6: `Relay-Ticket-Exchange`: $T^{\mathcal{C}}$ for $T^{\mathcal{R}}$ between relay $\mathcal{C}$ and bank $\mathcal{B}$ for service at relay $\mathcal{R}$. The arrows represent anonymous communication in Tor. The partially blind signature ($\textit{pbs-sign}(\cdot)$) and verification ($\textit{pbs-verify}(\cdot)$) are defined in [120].

```
Require:
  1: $\mathcal{C}$: generate random receipt $T_r^{\mathcal{R}}$
  2: $\mathcal{C}$: construct partial stub $\overline{T_s^{\mathcal{R}}} = \{\mathcal{R} \mid H(T_r^{\mathcal{R}})\}$
Ensure:
  3: $\mathcal{C} \rightarrow \mathcal{B}$: redeemable voucher $T^{\mathcal{C}}$
  4: $\mathcal{B}$: validate voucher $\textit{pbs-verify}(T^{\mathcal{C}})$
  5: $\mathcal{B}$: global ticket validity date-stamps $d = \{d_u \mid d_v \mid d_w\}$
  6: $\mathcal{B} \leftrightarrow \mathcal{C}$: partially blind-signature $\sigma = \textit{pbs-sign}(\textit{blind}(\overline{T_s^{\mathcal{R}}}) \mid d)$
  7: $\mathcal{C}$: construct full stub $T_s^{\mathcal{R}} = \{\overline{T_s^{\mathcal{R}}} \mid d \mid \sigma\}$
  8: $\mathcal{C}$: construct ticket $T^{R} = \{T_s^{\mathcal{R}} \mid T_r^{\mathcal{R}}\}$
```

Client Ticket Exchange

Since a client might obtain tickets for a relay who is offline for the duration of the ticket's spending interval, a client may exchange a ticket for another bound to a new relay. The `Client-Ticket-Exchange` protocol (not shown) is essentially identical to Algorithm 6, except that on line 4 the bank checks that the ticket is in the correct interval for client-exchange, but does not (and cannot) check for the identity of the client in the ticket.

Incorporating Tickets into the Tor Protocol

Our ticket construction from Section 7.3.1 enables us to easily embed ticket stubs and receipts in Tor messages (i.e. cells). As shown in Algorithm 7, `BRAIDS-Communication`, the client constructs the Tor message such that each relay on the path receives its own ticket stub and the receipt for the previous-hop in the path. There are two exceptions: the first-hop relay does not send a receipt to the client, and the last-hop relay receives a complete ticket (there is no next-hop relay).

We must include accounting mechanisms not only for the forward path from client to server, but also the reverse path from server to client due to asymmetric bandwidth requirements (e.g. streaming media). Since the reverse path cannot be paid by the server, clients pre-pay circuits (several cells can be transferred for each ticket) and relays

Algorithm 7: `BRAIDS-Communication`: Message M from Client $\mathcal{C}$ to server $\mathcal{S}$ through relays $\mathcal{R}_1$, $\mathcal{R}_2$, $\mathcal{R}_3$.

```
Require:
 1: $\mathcal{C}$ obtains tickets $T^{\mathcal{R}_i} = \{T_s^{\mathcal{R}_i} \mid T_r^{\mathcal{R}_i}\}$, for $i \in [1,3]$
 2: $M_{\mathcal{C}\to\mathcal{R}_3} = E_{K_{\mathcal{C}\mathcal{R}_3}}\{T_r^{\mathcal{R}_2} \mid T_s^{\mathcal{R}_3} \mid T_r^{\mathcal{R}_3} \mid \mathcal{S} \mid M_{\mathcal{C}\to\mathcal{S}}\}$
 3: $M_{\mathcal{C}\to\mathcal{R}_2} = E_{K_{\mathcal{C}\mathcal{R}_2}}\{T_r^{\mathcal{R}_1} \mid T_s^{\mathcal{R}_2} \mid \mathcal{R}_3 \mid M_{\mathcal{C}\to\mathcal{R}_3}\}$
 4: $M_{\mathcal{C}\to\mathcal{R}_1} = E_{K_{\mathcal{C}\mathcal{R}_1}}\{T_s^{\mathcal{R}_1} \mid \mathcal{R}_2 \mid M_{\mathcal{C}\to\mathcal{R}_2}\}$
Ensure:
 5: $\mathcal{C} \to \mathcal{R}_1 : M_{\mathcal{C}\to\mathcal{R}_1}$
 6: $\mathcal{R}_1$ : verify $T_s^{\mathcal{R}_1}$
 7: $\mathcal{R}_1 \to \mathcal{R}_2 : E_{K_{\mathcal{R}_1\mathcal{R}_2}}\{M_{\mathcal{C}\to\mathcal{R}_2}\}$
 8: $\mathcal{R}_2$ : verify $T_s^{\mathcal{R}_2}$
 9: $\mathcal{R}_2 \to \mathcal{R}_1 : E_{K_{\mathcal{R}_1\mathcal{R}_2}}\{T_r^{\mathcal{R}_1}\}$
10: $\mathcal{R}_2 \to \mathcal{R}_3 : E_{K_{\mathcal{R}_2\mathcal{R}_3}}\{M_{\mathcal{C}\to\mathcal{R}_3}\}$
11: $\mathcal{R}_3$ : verify $T_s^{\mathcal{R}_3}$
12: $\mathcal{R}_3 \to \mathcal{R}_2 : E_{K_{\mathcal{R}_2\mathcal{R}_3}}\{T_r^{\mathcal{R}_2}\}$
13: $\mathcal{R}_3 \to \mathcal{S} : M_{\mathcal{C}\to\mathcal{S}}$
```

notify clients when their paid balance expires. Clients embed tickets in outgoing cells using Algorithm 7, which distributes tickets to each relay in the circuit. A relay lowers a circuit's priority when not paid and restores it after new tickets arrive. Since scheduling decisions are made locally and independently, clients may choose to pay a subset of relays in the circuit without affecting scheduling decisions made by other relays.

Relays drop circuits upon detection of malicious activity, including forged tickets, and will only forward messages if a receipt is returned by the next-hop relay. Each relay is encouraged to participate faithfully to continue accumulating tickets since malicious activity stops the flow of tickets for all relays in a dropped circuit.

Double Spending

We have shown that relay-specific payments eliminates the trade-off between double spending prevention [121] and information leakage, suffered by PAR [113], since tickets can be verified by the relay without a third party. Anonymous payments are appropriate in Tor since they protect the identity and privacy of the user.

7.3.3 Randomized Ticket Distribution

A major problem with the gold star incentive scheme [47] is that gold star relays can be distinguished from normal relays, since their gold star status appears in the public directory. This reduces their anonymity set – an adversary can be confident that if a client is receiving gold star service, that client also runs a relay.

To mitigate this problem, we assign each client *ticket distribution agents* – guard nodes that assist in distributing free tickets to all clients. We note that each Tor client already uses a small set of guard nodes from which a circuit entry node is selected to limit client identity (IP address) exposure to malicious entry nodes. Each agent distributes tickets from the bank to clients in proportion to the bandwidth it provides as a relay: the client will create a secure connection tunneled through an agent to invoke the ticket distribution protocol (similar to `Relay-Ticket-Exchange` – Algorithm 6). Agents frustrate a *Sybil attack* [122], where clients join multiple nodes to the system to increase free ticket income, by limiting tickets distributed to each client's IP address.

Distribution Requirements

We require several properties as agents distribute tickets: nearly all clients should obtain tickets to remain indistinguishable from relays when spending; the algorithm that assigns clients to agents should not leak the client's identity to prevent an adversary from using a predecessor attack to infer a client from its agents; a client should obtain more tokens by becoming a relay than by cheating to maintain relay incentives; and a client's set of agents should be consistent over roughly the same period as they would if used as regular guards for stability.

Agent Assignment

Each client uses Algorithm 8 to determine which guard nodes it can use as distribution agents. A fundamental part of the protocol is the *hash–bandwidth test* for a guard $\mathcal{G}$. The test is true if the result of a cryptographic hash is less than the fraction of total guard bandwidth provided by $\mathcal{G}$ ([43] describes secure bandwidth measurements).

A client uses hash–bandwidth tests to walk through the guards while constructing a set of signature chains such that each chain can be verified as a correct chain for the

Algorithm 8: `Agent-Assign`: Clients verifiably compute their assigned ticket distribution agents by creating signature chains.

```
 1: $sig\_chains = [((clientIP, 0))]$
 2: for $i = 1 \rightarrow walk\_length$ do
 3:   $next\_step = [\,]$
 4:   for $chain \in sig\_chains$ do
 5:     $link = last(chain)$
 6:     for $\mathcal{G} \in guards$ do
 7:       if $hash(link \mid \mathcal{G}.pub\_key) < bandwidth\_frac(\mathcal{G})$ then
 8:         $sig = get\_sig(\mathcal{G}, clientIP, i)$
 9:         $next\_step.append(chain + \mathcal{G}.pub\_key + sig)$
10:       end if
11:     end for
12:   end for
13:   $sig\_chains = next\_step$
14:   if $sig\_chains = [\,]$ then
15:     $abort()$ // found no valid chains of proper length
16:   end if
17: end for
```

client. After completing Algorithm 8, each constructed chain is then used as input to a final round of hash–bandwidth tests (one for each guard): every guard that passes this final test is assigned as a distribution agent for the client.

A client builds signature chains as follows. On line 1, a client initializes a signature chain with its IP address and 0 as the current step in the walk. The client then initiates a pseudo-random walk: any guard $\mathcal{G}$ that passes the hash–bandwidth test is a valid next step, using the previous link in the chain and $\mathcal{G}$'s public key as input to the hash function (line 7). If $\mathcal{G}$ passes the test, it will return a signature to extend the chain and the walk (line 8-9). Although not shown on line 8, the client also sends the previous signature in the chain to prove to the guard that the signature request is valid (and not a waste of resources). Note that each step of a walk may break or fork a signature chain, hence the number of parallel walks performed, depending on the number of guards that pass the hash–bandwidth test. A client will continue extending its signature chains until it has walked *walk_length* steps, or has no next steps for any walk.

If Algorithm 8 does not terminate via *abort*(), then each chain in the list of *sig_chains* is a verifiable (but not public) token that attests the correctness of the assignment without leaking the client's IP address. The client uses each constructed chain to find an agent: a guard $\mathcal{A}$ that passes another hash–bandwidth test using the chain and $\mathcal{A}$'s public key as input to the hash function.

If Algorithm 8 terminates via *abort*(), then the client does not have any valid agents. However, to control the expected and median number of agents per client, we may add an adjustable parameter λ to the bandwidth fraction of each guard node. Probabilistic bounds show that the probability of having k agents will decrease exponentially in k, making it infeasible for an adversary to gain a large advantage by, e.g. manipulating the public keys of some agents.

Algorithm 8 requires that clients compute hashes for every guard, but is advantageous since it does not require re-assignment when agents churn and it load-balances distribution tasks among agents. Although Tor directory servers measure bandwidth [123], we require a secure bandwidth measurement technique such as [43]: the bandwidth values listed in the consensus become a security parameter since they determine the outcome of the hash–bandwidth tests, the number of agents assigned to a client, and therefore the total number of free tickets a client may receive.

Longer walks increase security since an adversary must compromise $walk\_length$ nodes to manipulate a signature chain. We suggest using $walk\_length = 3$ so that an adversary has a higher probability of compromising a circuit than compromising a walk: in the random oracle model, and assuming a deterministic signature scheme (like RSA+FDH), predicting (better than random guessing) whether a given guard is a valid agent for a client reduces to producing valid signatures for all $walk\_length$ signatures in the chain. With a walk length of 3, an adversary would not only need to control the final relay in the chain, but also either the first or second relay to obtain all three signatures. The fraction of nodes' agents that can be guessed in this way is on the same order of magnitude as the fraction of circuits that can be compromised by end-to-end attacks. We simulated agent assignment and found that using $walk\_length = 3$ and $\lambda = 0$ results in a median of one agent per node (details omitted for space reasons). Note that clients may use unassigned guards, but guards will not allow unauthorized clients form collect free tickets from them.

Since agents limit distribution to unique IP addresses, users behind NAT boxes will compete for handouts and aggregate performance for NAT users will suffer. Note that if IPv6 is universally adopted, the distribution scheme will require modification since each client can generate several IPv6 addresses [124]. We accept an adversary capable of joining multiple IPv4 nodes since it increases Tor's anonymity set.

Agent Collusion

Using guards as distribution agents introduces a chance for collusion. An adversary could join a relay to Tor, become a guard, and distribute tickets to a colluding client. To mitigate this problem, the bank will only allow an agent to distribute tickets if that agent has e.g. obtained the "stable" flag in Tor [125]. An agent cannot cheat until it has contributed significant resources.

The bank also limits distribution to each agent. Bandwidth measurements may be used to estimate the number of clients an agent is servicing, and the number of tickets the agent is allowed to distribute. Since agents are also guard nodes and ticket distribution is based on contributed bandwidth, the number of tickets they distribute directly correlates with the number of tickets they earn by relaying traffic. This can be used to bound the advantage agents gain by not honestly distributing tickets to clients.

Suppose agent $\mathcal{A}$ has bandwidth fraction b. Each agent has two non-agent guard nodes, and $\mathcal{A}$ receives $\frac{1}{3}$ of the tickets they spend when they select $\mathcal{A}$ as their entry node to Tor. Each selection occurs with probability b, so $\mathcal{A}$ receives $\frac{2 \cdot b}{3}$ of the tickets just by being a guard. If $\mathcal{A}$ additionally keeps the tickets $\mathcal{A}$ is supposed to distribute, the most tickets $\mathcal{A}$ can receive is $\frac{5 \cdot b}{3}$, about 2.5 times as many tickets. This is the worst-case: tickets re-spent by relays will lower this bound. Future work should consider auditing agents' ticket distribution to detect dishonesty.

Ticket Economy

Our ticket distribution strategy continuously introduces new tickets into the system that will eventually be exchanged at the bank. Continuous ticket exchanges impose a bandwidth constraint on the bank (see Section 7.4.1). Therefore, we must bound the total number of tickets that exist in the system at any one time in order to allow the bank to handle all exchanges that may occur during an exchange interval.

To bound the total number of tickets in the system, the bank imposes a *ticket tax* on users when exchanging tickets. The tax rate is adjustable based on the bank's bandwidth constraints and estimate of the total number of tickets currently in the system. The bank's estimate considers the number of tickets exchanged during previous exchange intervals (tickets not exchanged expire automatically). In practice, the bank can probabilistically fail each ticket exchange to reach the desired tax rate, but this consumes bandwidth resources for tickets that will be taxed. Alternatively, the bank could reveal random numbers that represent a hash output range during every exchange period, and tickets whose hash value falls in this range can be considered taxed and invalid. Then clients can discover which of their tickets have been taxed without contacting the bank. The anonymity implications involved with taxing and bounding tickets are discussed in detail below in Section 7.4.2.

7.3.4 Differentiated Service

BRAIDS employs differentiated services[3] and a scheduler based on the proportional differentiation model introduced by Dovrolis *et al.* [34, 109, 110, 87]. The model states that performance for each service class (in terms of measurable metrics like queuing delay) should be relatively proportional to parameters configured by the network operator. Let $q_i(t, t+\tau)$ be a performance metric measured during the interval $(t, t+\tau)$ for monitoring time scale τ. The proportional differentiation model creates quality differentiation parameters c_i for each class of service i and introduces constraints such that:

$$\frac{q_i(t, t+\tau)}{q_j(t, t+\tau)} = \frac{c_i}{c_j} \tag{7.2}$$

where $c_1 < c_2 < \ldots < c_n$. We write the delay ratio between these classes as $c_1 : c_2 : \ldots : c_n$. The performance metric under consideration should always maintain the proportions defined by the quality differentiation parameters, during any monitoring timescale.

We define the performance metric q_i to be the queuing delay of class i; the delay parameters between each class are adjustable. Dovrolis *et al.* contribute schedulers that approximate proportional delay differentiation under heavy loads. BRAIDS utilizes the Hybrid Proportional Delay (HPD) scheduler, which is a combination of the Waiting

[3]We also explored proportional delay differentiation, as is used in BRAIDS, in Chapter 5 (Section 5.3)

Time Priority (WTP) and the Proportional Average Delay (PAD) schedulers. Each Tor cell is time-stamped upon arrival at the relay and placed in the queue associated with the cell's class of service. When the relay makes a scheduling decision at time t, WTP computes the priority of only the cells at the head of each class i's queue as $p'_i(t) = \frac{w_i(t)}{c_i}$, where $w_i(t)$ is the waiting time of the cell computed using the time-stamp from above. PAD computes class priorities as $p''_i(t) = \frac{a_i(t)}{c_i}$, where $a_i(t)$ is the total average delay incurred by service class i before time t. HPD weights these priorities as $p_i(t) = p'_i(t) \cdot (1 - f) + p''_i(t) \cdot f$, where f is an adjustable fraction. The cell with the highest computed priority $p_i(t)$ is scheduled. In BRAIDS, the HPD scheduler computes at most six priorities for each scheduling decision.

HPD allows us to differentiate performance of paying and non-paying clients by adjusting the c_i parameters. We then divide client traffic into three distinct service classes: (1) *Low-latency* for web browsing clients, (2) *High-throughput* for file sharing clients, and (3) *Normal* for non-paying clients. These classes will be proportionately delayed as low-latency : high-throughput : normal.

Low-latency Service

Users who wish to browse the web typically want fast response but not high throughput. Therefore, we schedule low-latency traffic with the highest priority. We rate-limit low-latency traffic for each circuit to prevent users from sending high traffic loads and overwhelming the low-latency class; traffic exceeding a threshold limit over a monitoring timescale will be demoted to the normal class. We suggest a threshold equal to the number of free tickets each user receives during a spending interval (discussed in Section 7.4.1). Throttling is necessary to prevent high-throughput clients from "abusing the pipe" for web users, which is currently a well-known problem in Tor [28].

High-throughput Service

Conversely, clients with high throughput requirements (e.g. BitTorrent users) tolerate higher-latency service. Therefore, we increase scheduling delays relative to the low-latency class but do not throttle traffic. As a result, high-throughput traffic has a diminishing effect on low-latency traffic.

Normal Service

Since not all users will be able or willing to deploy relay services, we do not *require* clients to make payments in order to use BRAIDS. Instead, clients who have expended their free ticket allowance, or choose not to pay for service, receive both the lowest priority and, in turn, the highest scheduling delays.

Differentiating service results in two interesting consequences: it provides incentives to run relays, since users in higher service classes receive lower delay; and it allows for incremental deployability by placing traffic from legacy clients in the normal service class. Note that the extent of the performance gain between service classes depends on the chosen delay parameters.

7.4 Analysis and Discussion

7.4.1 Parameter Selection

Ticket Validity Intervals

Recall that ticket validity intervals [-,d_u), [d_u,d_v), and [d_v,d_w) are global uniform intervals in which tickets may be spent and exchanged and are broadcasted by the bank (see Section 7.3.1). We now explore the frequency and relative timing of each interval.

To prevent unspendable ticket periods, tickets that are received in spending interval i are exchanged in spending interval $i + 1$ (exchange interval i overlaps spending interval $i + 1$). Time in each exchange interval is shared between a relay-exchange period and a client-exchange period such that the fraction of time allotted for relay exchange corresponds with the expected fraction of tickets relays possess (which the bank can estimate based on exchanges in previous intervals).

Using the interval strategy just described, the bank will only exchange half of all tickets in the system during every spending interval and users can only spend half of their tickets at one time. Following this approach, tickets received in spending interval i are exchanged in spending interval $i + 1$ and spent in spending interval $i + 2$. All tickets not exchanged during an exchange interval will expire, so if relays are offline for the duration of an exchange interval, they will lose roughly half of their tickets.

Longer spending and exchange intervals means relays must wait longer to use tickets, but shorter intervals means tickets expire faster. We suggest a compromise of 24 hour spending and exchange intervals, noting that future work should consider a further exploration of exchange intervals.

Ticket Worth

Recall that several cells may be transferred through Tor for each ticket. The number of cells transferred for each ticket has an important impact on the bank's CPU and bandwidth consumption. Since we limit the amount of data users can download for free, higher ticket worth means the bank has to exchange fewer tickets, reducing both CPU and bandwidth requirements. However, users then have fewer tickets overall which reduces the number of independent circuits that can be paid simultaneously. We suggest that users receive 3 tickets every 10 minutes so they may utilize a prioritized circuit at any time. We note that in practice these tickets will likely be freely distributed in batches at a higher time granularity (e.g. every hour).

Cryptographic and Bandwidth Costs

Each relay must perform a SHA1 hash and an AES encryption/decryption for each cell it transfers. BRAIDS introduces an additional task – verification of a ticket. We implemented the partially blind signature scheme of Abe *et al.* [120] using GMP [126] for arbitrary precision arithmetic. We measure both the amount of time a bank spends producing a signature, and the amount of time a relay spends verifying a single ticket. We also compute the time to perform the SHA1 and AES operations required by Tor.

Table 7.1 shows the results of our Linux benchmarks on 3 GHz AMD 64 Athlon X2 6000 and 2.67 GHz Intel Core 2 Duo 6750 CPUs. We report the mean, median, and standard deviation of times, in microseconds, for each operation described above. As expected, a signature verification takes significantly longer than AES and SHA1 operations currently performed by relays. However, the value of each ticket can be selected such that a ticket need not be sent for every cell and expensive ticket verification costs can be amortized. An appropriate value would result in a greater cost for AES and SHA1 than for verifications to prevent the signature scheme from becoming a bottleneck. Our benchmarks suggest a single ticket be worth 128 KB of data so that a verification need only be performed for every 256 cells.

Table 7.1: Cryptographic time per cell for Tor relays compared with BRAIDS PBS verifications, in microseconds. Also shown is the bank's time per signature.

		Mean	Median	Std. D.
AMD Athlon (3 GHz)	AES+SHA1	9.139	8.616	2.493
	PBS verify	531.287	530.885	8.342
	PBS bank	413.244	412.069	7.297
Intel Core2 (2.67 GHz)	AES+SHA1	6.226	5.859	1.307
	PBS verify	1496.813	1496.613	10.844
	PBS bank	1193.233	1192.782	9.472

Given our Intel benchmarks, a relay performs roughly 666 verifications per second while the bank may perform over 833 signatures per second per processor core. Each relay may therefore upload at a rate of 666 Mb/s while streamlining verification procedures, and the bank may sustain an aggregate 833 Mb/s of prioritized traffic through Tor. Recall that the bank is offline and may be distributed among multiple physical machines for additional computational processing resources.

To compute bandwidth costs, suppose a user receives η free tickets during each spending interval. Not including protocol overhead, which can be minimized by batching ticket exchanges, each ticket exchange consumes 488 bytes of bandwidth (the partially blind signature from Abe et al. [120] requires multiple messages between the client and the bank). In aggregate, the bank distributes $\eta \cdot \mu$ tickets per day, where μ is the total number of users receiving free tickets. Ticket exchanges are taxed such that after ρ spending intervals, $\eta \cdot \mu$ tickets are eliminated from the economy. The total number of tickets in the system is $\eta \cdot \mu \cdot \rho$ in expectation.

Since the spending and exchange intervals overlap, the bank will exchange and produce signatures for $\frac{\eta \cdot \mu \cdot \rho}{2}$ tickets every spending interval. If we assume a spending interval is 24 hours following our interval strategy from above, $\eta = 432$, $\mu = 100{,}000$, and $\rho = 20$, then the bank must sustain bandwidth loads of 20 Mb/s and perform 5,000 signatures per second, within reason of a multi-core CPU with a cryptographic accelerator.

7.4.2 Security Analysis

To measure the impact of BRAIDS on sender anonymity, we analyze information leakage in terms of an *anonymity probability distribution* [127, 128]. This analysis technique uses information-theoretic entropy [129] as a measure of information contained in a probability distribution. We define a discrete random variable I as a circuit initiator and compute a distribution of all potential initiators as a probability mass function $Pr(I = i) = p_i$ where p_i is the probability that a user i is the circuit initiator given the observations on the system. The entropy H of our distribution is:

$$entropy = H(I) = -\sum_{i=1}^{N} p_i \log_2(p_i) \tag{7.3}$$

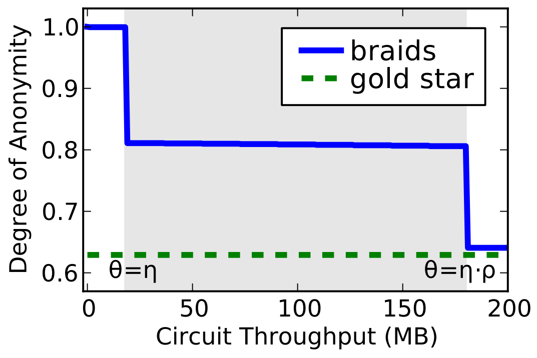
where p_i is the probability for user i taken from the distribution and N is the size of the anonymity set (the set of potential circuit initiators). The maximum entropy in the system H_M is computed as $H_M = \log_2(N)$. The *degree of anonymity* [127] quantifies information leakage and can then be defined as the fraction of total entropy obtained from the given distribution I:

$$degree\,of\,anonymity = \frac{H(I)}{H_M} \tag{7.4}$$

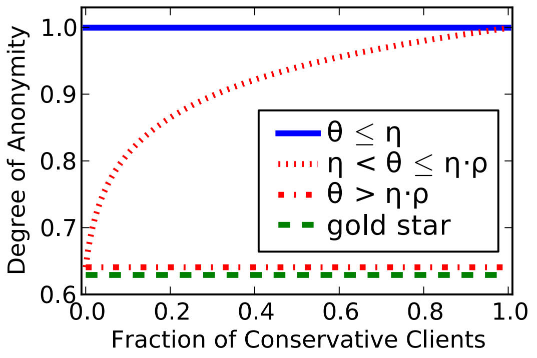
Distinguishability

To determine the effects of distinguishing clients from relays, we first assume that clients will fill one of two roles: a *liberal* client who spends tickets immediately by downloading web pages, and a *conservative* client who stores tickets until they can download a large file. Our liberal-conservative client model captures potential BRAIDS spending habits – in practice some clients will consistently spend most of their tickets while others will consistently underspend. We further assume that each relay in the system always has the desired number of tickets for any circuit it initiates to simplify analysis. We note that this is a coarse model as it is difficult to estimate users' spending habits.

While the tax rate ρ allows the bank to remove tickets from the system to keep ticket exchanges within manageable bandwidth bounds, it also potentially reduces anonymity for large downloads. If a user spends more tickets than is possible to collect only from



(a) Circuit Throughput Effect on Anonymity



(b) Conservative Clients' Effect on Anonymity

Figure 7.1: Anonymity is highest when the adversary observes fewer than $\theta = \eta$ tickets per circuit and lowest when more than $\theta = \eta{\cdot}\rho$ are observed. (a) In the shaded area, only $\frac{1}{10}$ of clients are conservative, collecting tickets longer than one spending interval. (b) Anonymity increases with conservative clients that contribute to adversarial uncertainty.

free distribution $(\eta \cdot \rho)$, an adversary can determine with high confidence that the circuit initiated from a relay by observing $\theta > \eta \cdot \rho$ tickets spent in a circuit. An adversary may additionally determine *which* relays can afford a given circuit by performing bandwidth measurements, since a relay's ticket income corresponds with the bandwidth it provides. However, in Section 7.4.1 we have suggested distributing enough free tickets to pay for general web browsing so that the majority of users will not spend over $\eta \cdot \rho$ tickets.

Discussion

To analyze our system, we obtain the growth rate of Tor relays from [125]. We estimate the client growth rate by analyzing how the number of client connections to a relay changes over a two month experiment [130]. We apply these rates and the estimated network size of 100,000 clients and 1,500 relays to find the total network size over time. From Section 7.4.1, each ticket is worth 128 KB of data transfer and we distribute $\eta = 432$ tickets per day. Tickets are taxed such that the system's ticket capacity is $\rho = 20$ cumulative days of tickets. The fraction of conservative clients is $\frac{1}{10}$, except where noted.

Figure 7.1 shows how the circuit throughput and fraction of conservative clients may affect the set of potential initiators (if an adversary can guess this fraction) and therefore the degree of anonymity BRAIDS provides. By observing $\theta < \eta$ tickets in a circuit, an adversary is unable to infer information about the circuit initiator since all liberal and conservative clients obtain η tickets during a spending interval. Observing $\theta > \eta \cdot \rho$ tickets means the circuit must have been initiated from a relay. For other observations, the degree of anonymity depends on the number of clients the adversary can eliminate from the potential initiator set.

The shaded area in Figure 7.1(a) represents anonymity when $\frac{1}{10}$ of clients are conservative. This fraction is an estimate: it is difficult to determine how users will spend in practice. We explore the effects of varying the conservative client fraction in Figure 7.1(b). Since conservative clients represent adversarial uncertainty, we find that having more conservative clients has a positive effect on anonymity. In all cases, anonymity is higher in BRAIDS than the gold star scheme where only the fastest $\frac{7}{8}$ of relays are potential prioritized-traffic initiators. For highest anonymity, clients should spend less than η tickets for prioritized traffic in each spending interval.

7.5 Simulation and Results

We simulate BRAIDS and Tor to compare performance and illustrate how effective our system is at encouraging users to run relays. Below we describe our simulator, experiments, and results.

7.5.1 Simulator

We built a discrete-event-based simulator that models the Tor network. Within the first ten minutes of an experiment, all Tor clients start one of the applications described below and begin generating data. Each client builds circuits following Tor's path selection protocol [131], and refreshes each circuit after ten minutes, building a new one when the next request is made. We now describe our client applications.

Web Clients

Each web client (WC) generates traffic by making a top-level page request and waiting for a response from the server. After receiving a response, the WC makes several additional parallel requests for objects embedded in the page (e.g. images). After receiving all embedded object responses, the WC waits for a period of time before downloading another page. We record the time required to download the entire page, including all embedded objects. The period between the initiation of the top level request until the reception of the final embedded object simulates the time required to render an entire page in a user's browser. Distributions for all request and response sizes, the number of embedded objects per page, and the time between page requests are taken from the web traffic study conducted by Hernandez-Campos *et al.* [58].

File Sharing Clients

Each file sharing client (FSC) simulates a BitTorrent-like protocol by continuously generating data to five random peers through the Tor network. Every thirty seconds the FSC will replace its slowest connection with a new peer and a new circuit, simulating BitTorrent's "optimistic unchoke" algorithm [132]. Each FSC exchanges blocks by sending a 32 KB request for a 32 KB reply and immediately sending another request upon receiving a reply. We measure the time to exchange each block.

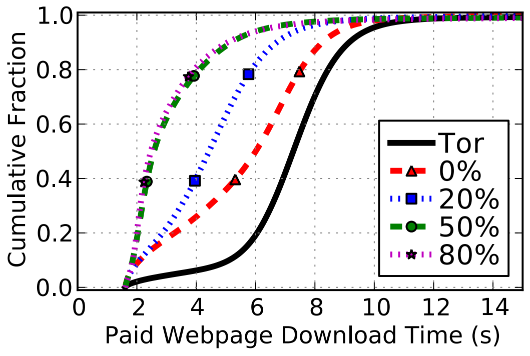
File Sharing Relays

A file sharing relay (FSR) implements the same algorithm as a FSC with the following deviation: FSRs contribute a fraction of their total upstream bandwidth to Tor while using the remaining bandwidth for their own file transfers. The bandwidth contributed by FSRs supplies them with additional income not received by FSCs.

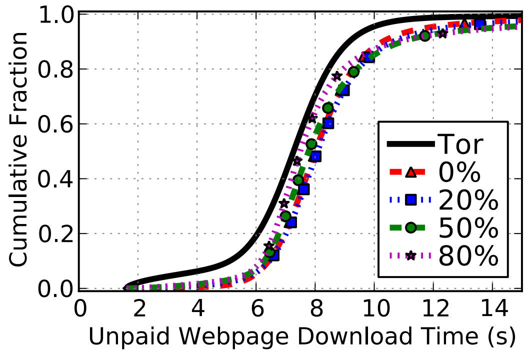
We simulate every cell generated by each client and sent through the Tor network. Tor nodes schedule outgoing cells using an exponential weighted moving average (EWMA) scheduler [33], while BRAIDS nodes use the HPD scheduler (see Section 7.3.4). To bootstrap the economy, tickets are distributed to all clients and all relays at the beginning of each simulation.

7.5.2 Experimental Parameters

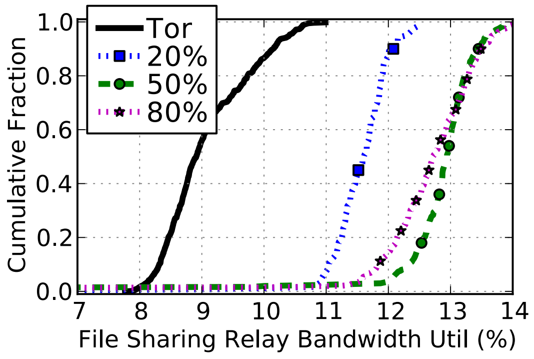
Our simulated network consists of 19,400 web clients, 300 Tor relays, 2,000 servers, and 600 file sharing nodes. Our web and file sharing nodes are given consumer-class connections of 12 Mb/s downstream 1.3 Mb/s upstream bandwidth, and 24 Mb/s downstream



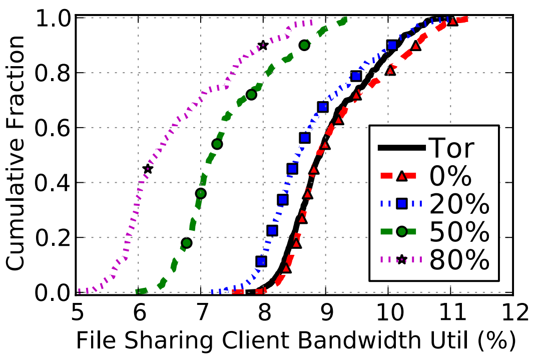
(a) Prioritized Web Clients



(b) Normal Web Clients



(c) File Sharing Relays



(d) File Sharing Clients

Figure 7.2: BRAIDS and Tor simulated performance with a varying percentage of File Sharing Clients converting to File Sharing Relays. Webpage download time for (a) paid, low-latency service, and (b) unpaid, normal service. Bandwidth utilization for (c) file-sharing relays, and (d) file-sharing clients.

3.5 Mb/s upstream bandwidth, respectively.[4] File sharing relays draw contributed bandwidth amounts from the Tor network consensus [133] repeatedly until obtaining a value below their upstream capacity.[5] Altruistic relays are given symmetric upstream and downstream capacities drawn from the bandwidth distribution reported in the consensus, clipped at 20 MB following standard Tor procedure [131]. Servers are given unlimited bandwidth and we impose no processing delay on any node. Network latency between every hop is set to 100 ms, and we do not account for membership churn or congestion control in our simulator since it will have a similar effect on both Tor and BRAIDS performance. Our simulation runs for 60 virtual minutes.

We run multiple BRAIDS and Tor experiments with the above parameters, using 1 : 64 : 4096 as the HPD scheduler's delay parameters corresponding to the service class ratio low-latency : high-throughput : normal, and HPD fraction $f = 0.875$ (see Section 7.3.4). Since we are interested in the incentives our system provides for running a relay, we vary only the fraction of 600 nodes that are FSCs as opposed to FSRs. This will allow us to determine how a user's performance changes by serving as a relay. The load on the network is unchanged between all experiments. Our simulator closely approximates empirical Tor traffic loads gathered by McCoy *et al.* [28].

7.5.3 Results

In BRAIDS, the low-latency service class achieves a significant reduction in download time compared with Tor, and download times improve as more FSCs convert to FSRs (Figure 7.2(a)). Since web browsers transfer small amounts of data in most cases, improvements in download times are noticed even with few new relays. The similarity in download time when 50% and 80% of FSCs change to relays suggests that these nodes have reached a lower bound. We note that the best possible download time is 1.6 seconds, since all web clients must make at least one top-level and one embedded object request, resulting in sixteen 100 ms hops. The normal service class webpage download time is longer than in Tor, and performance slightly declines as more file sharing users move to the high-throughput class since normal data is proportionally delayed sixty-four times as long as high-throughput data (Figure 7.2(b)). Unpaid traffic performance

[4]ADSL Standard ITU G.992.1 Annex A, ADSL2+ ITU G.992.5 Annex M.

[5]The consensus document was obtained on January 12, 2010 between 18-19:00:00 CST.

is best when 80% of the FSCs convert to relays since clients can take advantage of a significant increase in available bandwidth. These results are outstanding – download time for normal web traffic does not unusably degrade from performance achieved in Tor, and running a relay will provide a definite performance boost over those who choose to remain client-only.

BRAIDS FSRs not only receive an improvement in bandwidth utilization over Tor, but can also achieve up to approximately 90% better utilization of their bandwidth compared with BRAIDS FSCs that do not run a relay, even while contributing a fraction of their bandwidth to Tor. Figure 7.2(c) shows that FSRs performance increases as more nodes convert to relays. However, since the newly available bandwidth is also consumed by WCs, relays realize only incremental improvements as the fraction of converting relays increases. Figure 7.2(d) shows that as more filesharers convert to relays, performance for FSCs degrades. This happens mostly because a large amount of data from FSRs is receiving priority over data from FSCs, and the newly available bandwidth is being consumed by the low-latency and high-priority service classes. For all conversion rates, FSRs achieve considerably better performance than FSCs.

Overall, our results strongly indicate that BRAIDS users can increase the performance of both interactive and non-interactive traffic by starting a relay and contributing bandwidth to Tor. Therefore, if users want to run BitTorrent or similar file sharing protocols using BRAIDS, they should run a relay to achieve maximum performance. This, in turn, will have a positive impact on the entire network since there will be more bandwidth available for other Tor clients.

7.6 Summary

In this chapter we introduced BRAIDS as a set of practical mechanisms that encourages users to run Tor relays. We employ completely client-anonymous relay-specific tickets that allow Tor clients and relays to achieve increased performance while preventing the double-spending problem. Relays differentiate service into three classes, allowing them to prioritize traffic and mitigate the negative effect file sharing users have on Tor, without significantly reducing bandwidth utilization for file sharing clients.

Chapter 8

LIRA: Lightweight Incentivized Routing for Anonymity

8.1 Introduction

Tor relays are run by volunteers who altruistically contribute bandwidth and computational resources to the network. As a result, Tor is usable even by those unable or unwilling to contribute because they, e.g., have slow connections or are behind restrictive firewalls. Unfortunately, network capacity is limited to these altruistic contributions and has increased sublinearly to Tor's popularity. In Tor's current resource model, its popularity harms its usability and performance, and may therefore have a significant negative impact on its users' anonymity [25, 26]. The Tor Project [12] has enumerated many performance problems they have recognized and are actively pursuing designs that improve the network in this regard [24]. Recent work has focused on reducing the existing load on the network [94, 134] and improving the utilization of the existing relay resources [33, 40, 97], but bolstering capacity while at the same time encouraging scalability remains a challenging open problem.

Various responses to this capacity and scalability problem have been considered. Thus far Tor has relied on community support to provide much-needed boosts to its capacity. For example, Torservers.net [135] is a registered German non-profit organization that uses donations to purchase or rent high-bandwidth servers for the public Tor network. Similarly, the Electronic Frontier Foundation ran a "Tor Challenge" in which they encouraged people to set up relays and listed the names of those who chose to be acknowledged for doing so [136]. Unfortunately, the support is limited and inadequate for Tor to scale to millions of simultaneous users while remaining usable. Currently Tor is initiating direct funding of relays using government funding it receives for this purpose. As noted in the blog post announcement, this raises numerous questions, such as the impact on diversity of the infrastructure [137]. Another unknown is the sustainability of any resulting capacity increase if this direct funding ceases.

A more scalable way to increase capacity is to require all users to contribute in a peer-to-peer fashion. However, not only would it be difficult to force users to comply, this would also turn away some of those most in need of its protections due to an inability to contribute. Further, combined with a potential lack of user interest in operating and maintaining servers, this strategy may produce undesirable low bandwidth or unstable relays that increase network bottlenecks and may actually harm performance [43].

Numerous proposals to recruit new relays using incentives have appeared in the literature. Although the incentive approach is promising, past designs have thus far been plagued with anonymity or efficiency problems. Both the "gold star" scheme [47] and Tortoise [94] have serious anonymity problems that allow relays' traffic to be identified, while PAR [113], XPAY [114], and BRAIDS (Chapter 7 [46]) do not not scale well due to inefficient protocols. Our goal in this work is to design and evaluate a system that combines *strong anonymity* with *scalable efficiency*.

8.1.1 Lightweight Incentivized Routing for Anonymity

We present LIRA, a unique and scalable approach to creating incentives for users to contribute computational and bandwidth resources to Tor. Proportionally differentiated services [34] are the foundation for incentives: users who choose to run relays will be able to proportionally increase their performance relative to those not contributing. Further, relays may contribute more resources to increase the amount of their traffic that gets prioritized, leading to greater network capacity and performance improvements for everyone. At the same time, LIRA frustrates the adversary's ability to utilize traffic priority as a distinguisher of client-initiated and relay-initiated circuits.

LIRA produces incentives with a novel cryptographic *lottery* design together with a new circuit scheduling algorithm that prioritizes traffic from those winning the lottery. To play the relay lotteries, clients send a *ticket* to each relay in each circuit built in LIRA. Clients generate random number *guesses* to produce tickets locally, each of which will be a *winner* for a relay lottery with a tunable probability. Relays contributing resources may collect anonymous *coins* proportional to their contributions, and exchange the coins for guaranteed winners to relay lotteries of their choosing. Relays differentiate performance by prioritizing traffic for winning circuits.

LIRA maintains anonymity. An adversary in LIRA is unable to distinguish relays' purchased winners from clients' guessed winners, whereas an adversary in the "gold star" [47] and Tortoise [94] incentive designs can determine that traffic initiated from relays with absolute certainty. LIRA provides tunable anonymity: increasing the probability that a guessed ticket is a winner reduces the adversary's certainty about the traffic source.

LIRA is lightweight. Previous schemes either require that an online trusted third party participates in routing in order to prevent double spending, as in PAR [113] and XPAY [114], or that the third party distributes tickets to all relays *and all clients*, as in BRAIDS [46]. Neither of these approaches scales well to millions of simultaneous users. LIRA is scalable because purchased tickets are not managed for clients, but only for the orders of magnitude smaller set of relays, and there is no spending transaction when circuits are built.

8.1.2 Contributions

This work's major contributions may be summarized as follows:

- A unique and novel cryptographic lottery approach to providing incentives to run Tor relays that combines strong anonymity with scalable efficiency
- A new Tor circuit scheduler that produces performance incentives through proportional throughput differentiation
- A detailed efficiency, anonymity, and incentive analysis and a comparison to BRAIDS, the state-of-the-art Tor incentive design
- A prototype implementation and experimental validation that LIRA provides incentives to contribute

8.1.3 Outline

The rest of this chapter is outlined as follows. Section 8.2 provides details about the network, our threat model, and our objectives. LIRA's technical design is given in Section 8.3, while Section 8.4 analyzes LIRA's efficiency, anonymity, and incentives. Our prototype and experimental evaluation are described in Section 8.5, and Section 8.6 concludes.

8.2 Preliminaries

We now discuss specific details about the deployed Tor network that LIRA's design considers and describe the circuit building protocol to facilitate an understanding of how we will propose to modify it. We also introduce a bank as an additional service that will be utilized in LIRA's design, specify our adversarial threat model, and clarify the objectives of our system. Though LIRA could be applied to various anonymous communication systems, our exposition will focus on the Tor [11] onion routing network.

8.2.1 Onion-Routing Network

The most popular instantiation of onion routing [10], the Tor overlay network includes a *directory service* that publishes information about the available relays. Using the directory information, clients build three-hop circuits that begin with one of a small set of entry guard relays and end with an exit relay willing to connect to the client's desired Internet service. A circuit is built through a *telescoping* process: an encrypted tunnel is first created to an entry guard, after which the tunnel is extended one relay at a time until the circuit is completely established at the exit relay. During this building process, the client negotiates an ephemeral key with each relay in the circuit using a Diffie-Hellman key exchange protocol. Once established, client TCP streams that conform to the exit relay's exit policy may be multiplexed over the circuit for ten minutes, after which the circuit will be marked as dirty and will not permit any new application connections. The circuit is destroyed once existing application connections are done using it. All data transferred over the circuit is packaged into uniform-sized *cells* and encrypted using the negotiated ephemeral keys.

A relay may be servicing several circuits at any given time. Every circuit that results in data exchange between any pair of relays is multiplexed over a single TCP onion-routing connection between the pair. Cells read from this connection are processed and placed in a scheduling queue before being switched onto the corresponding outgoing onion-routing connection to the next-hop relay.

Roughly 3000 geographically diverse Tor relays currently transfer a combined total of about 1700 MiB/s from an estimated 400,000 unique users per day [30]. We parameterize the onion-routing network size for design and analysis purposes as m onion routers and n unique users in a given time period. We also assume the existence of a new bank service B, assisting both in establishing valid lotteries with the relays and in assessing and rewarding contributions (see Section 8.3). We will assume that all entities can use the underlying communication network to send each other messages directly.

8.2.2 Adversary

We will consider the actions of both a malicious network adversary and an honest-but-curious (*i.e.* passive) bank. We use the standard network adversary for onion routing [138], which is *local* in that he can observe part of the network, is *active* in that he can perform computations, send messages, run onion routers, and act as a client. Although in this chapter we model the bank as a single entity, we expect the ultimate implementation of the bank to be similar to that of the directory service in the current public Tor network: multiple entities run the service and form a consensus on the authoritative documents. Therefore, we assume that the bank faithfully executes the protocol and only makes observations that are part of that protocol. In particular, he does not act as an onion router or as a client, only observes messages that are sent to him, and does not collude with the network adversary.

8.2.3 Objectives

Our goal is to provide incentive for anonymous-network users to run relays while preserving the desirable features of onion routing. Therefore, we will evaluate LIRA in terms of its functionality, its efficiency, the anonymity it provides, and the incentives it offers to those users who choose to run a relay.

We require that our system provide the functionality provided by onion routing. In particular, it should provide bidirectional, stream-oriented, low-latency communication between pairs of users. In addition, the responder of a stream should only need to run a standard transport protocol so users can communicate with destinations that aren't aware of or designed for anonymous communication protocols.

We also require that the efficiency of our system is comparable to onion routing. The success of Tor over alternative anonymous-communication protocols can be attributed in large part to its relatively low computational and communication costs. In particular, our protocol should have costs for each user that are proportional to amount of his anonymized traffic and for relays as a group that are proportional to the total amount of anonymized traffic. Moreover, we want the bank's resource requirements to be achievable under current Tor network conditions and to scale well with growth.

Our evaluation will consider *relationship anonymity* [138] in our system, that is, the

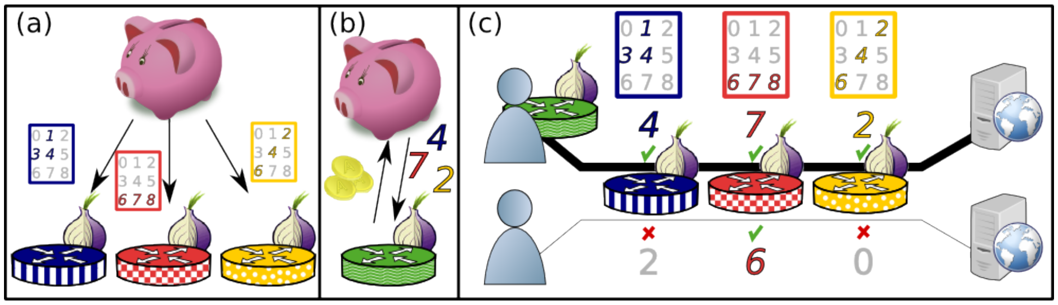


Figure 8.1: An overview of LIRA's design. (a) Relays coordinate with the bank to learn which tickets will be winners for their lottery. (b) Relays accumulate anonymous coins by contributing bandwidth to Tor, and may exchange them for guaranteed winners to other relay lotteries. (c) Clients send either guaranteed winners or guesses through their circuits. Relays proportionally differentiate throughput by prioritizing circuits that submitted winners to every circuit position.

extent to which users can be linked to their communication partners. We will measure this using the probability that an adversary assigns to a user communicating with his actual destinations. We will also evaluate the incentives provided by LIRA. As it is designed to improve throughput and latency for users running relays, we will measure this performance difference while also considering the extent to which a user can cheat and obtain these improvements without contributing.

8.3 Design

To achieve the objectives stated in Section 8.2, LIRA employs a cryptographic lottery and a relay circuit scheduler that prioritizes traffic for users who submit winning lottery tickets. A high level overview of LIRA's design and the interactions between these mechanisms is given in Figure 8.1. Through coordination with the bank, the relays receive information allowing them to recognize winning tickets to their own lottery (see Figure 8.1(a)). Over time, relays accumulate anonymous electronic coins from the bank by providing service to the network. These coins may be exchanged for guaranteed winning ticket values for a variety of relay lotteries (see Figure 8.1(b)). Clients without coins guess ticket values to produce them locally: their guesses will be winners with tunable probability. Tickets are passed to the relays through circuit control messages, and relays cannot distinguish a guessed winner from a guaranteed winner. Relays in every circuit position verify tickets and prioritize circuits of submitted winners by proportionally increasing their throughput (see Figure 8.1(c)). We now describe LIRA's design in further detail.

8.3.1 Setup

The bank will use RSA blind signatures [139], which allow it to sign a message without being able to link the signature with an earlier signing request. Let M be the RSA modulus of the bank, e be the public encryption exponent, and d be the corresponding private decryption exponent. Each relay r will need a public random value $x_r \in \mathbf{Z}_M^*$ associated with it. These values can be generated by the bank and distributed by the directory service. For each relay r, the bank computes its signature x_r^d and sends it to r. Finally, the system will use a full-domain hash function $H : \{0,1\}^* \rightarrow \{0,1\}^{\lambda/2}$, where λ denotes the security parameter for the system. We can use a cryptographic hash function such as SHA-1 for H.

8.3.2 Coin Distribution

LIRA rewards relays proportional to the amount of bandwidth they contribute. Since relays can not be trusted to self-report their bandwidth contributions, we determine each relay's contribution with a secure bandwidth measurement scheme such as EigenSpeed [140]. Using EigenSpeed, relays opportunistically measure and evaluate each other's contributions to form an accurate consensus of relay bandwidth that has been shown to be resistant to attacks by malicious groups of colluding nodes [43, 140]. The measurement process runs continuously while a consensus is formed periodically.[1]

The bank stores and tracks each relay's bandwidth contribution over time, keeping an account balance of contributed bytes and updating it with each new bandwidth contribution consensus. A relay may collect ℓ digital *coins* from the bank for every α bytes it has contributed, where ℓ is the circuit length ($\ell = 3$ in Tor). A coin is constructed using a blind signature [139] to prevent the bank from later linking the coin to a given relay (the final signature is unknown to the bank).

[1]Tor currently computes the directory consensus every hour, which could be amended to include the bandwidth contribution information.

Relays use their coins to purchase guaranteed winners from the bank (see Section 8.3.3). The bank *prevents* double spending of these by keeping a database of previously spent coins that it checks (and possibly updates) when it receives coins in a purchase request. The size of the database is bounded by a coin expiration period η. A blind signature simplifies the construction of the coin over some electronic cash schemes since we do not require double spending *detection* by a third party after a coin has been successfully spent.

Advantages of using coins in the manner outlined above include flexibility and transferability. Coins are *flexible* because relays may accumulate them during periods when they are not actively using Tor as a client, and they are valid as long as the bank exists and the coin has not expired. The expiration period η is set so that the bank can store and access a list of spent, unexpired coins and may be adjusted as the Tor network scales. Section 8.4 shows that currently in Tor 127.5 coins would be generated per second. If each coin is a 1024-bit signature, we can set $\eta = 28$ days, resulting in list of at most 4.60 GiB and fitting into a single machine's memory.

A coin is inherently *transferable* because it is not tied to a specific relay, allowing the possibility of a secondary economy to form around the purchase and sale of coins. In such an economy, clients who do not run relays can also obtain coins, improving anonymity by increasing the uncertainty of the sources of winning tickets.

We configure the ratio of the number of contributed bytes α to the number of prioritized bytes β received in return to $\alpha = (\ell + 1) \cdot \beta$. By requiring a contribution $\ell + 1$ times that of prioritized consumption, we account for transferring data through each of the ℓ relays in the circuit, and also ensure that new relays that join Tor will only increase its capacity.

8.3.3 Purchasing Guaranteed Winners

Relays will prioritize traffic on circuits for which winning lottery tickets are supplied. Winners will be determined using a relay-specific permutation that we define below (Eq. 8.3). Let the size of the permutation's input space be 2^λ, and let $g_r : [2^\lambda] \to [2^\lambda]$ be the permutation of relay r. The set of winners for r is $\{x : 1 \leq g_r(x) \leq p2^\lambda\}$, where $p \in [0, 1]$ is a system parameter. Thus a client that guesses an input x randomly will obtain a winner with probability p. To guarantee priority, a client can also use coins earned by providing service to the network to purchase winners.

Setting p presents a tradeoff between anonymity and incentives. If p is small, guessing a winner is unlikely. In this case, prioritized circuits are likely to be paid for and thus probably originate at a client also running a relay. If p is large, it is more likely that a circuit will be prioritized by chance, and there will be less reason to run a relay and earn priority. (We discuss this tradeoff in more detail in Section 8.4.) We adopt a setting of $p = n^{-1/(2\ell)}$. For the current Tor network, we estimate n to be 10000, and thus we would set $p = 10^{-2/3} \approx 0.22$.

The construction of the permutations g_r is designed to provide properties similar to those of a pseudorandom permutation (PRP), although they are technically somewhat different. In particular, the permutations will appear random to clients so that they *cannot produce winners with probability significantly greater than* p. Moreover, they are *efficiently invertible* so that the bank can sell winners by choosing outputs $1 \leq y \leq p2^\lambda$ and providing the corresponding input $g_r^{-1}(y)$. The construction also allows the purchase of winners for r while *hiding the identity of* r *and the winning number from the bank.*

If we didn't want to hide this information from the bank, we could easily implement the rest of this functionality by using a PRP such as AES. The bank could share different private keys with each of the relays, and the user would simply purchase a winner by presenting a coin and specifying a relay. We wish to keep the bank as oblivious as possible, and thus we use a more involved construction for the lottery permutations.

The essential ingredient of the construction is for the bank to use blind signatures to obliviously provide a relay-specific input to a certain pseudorandom function (PRF) (Eq. 8.1). We then use a construction similar to that of Luby and Rackoff [141] to convert the PRF into a PRP.

Private Evaluation of Pseudorandom Functions

The PRF we use is adapted from one suggested by De Cristofaro et al. [142] that can be computed obliviously.[2] Our construction doesn't provide full obliviousness with respect to the bank, but it will provide privacy assuming that the bank does not collude with

[2]The PRF they suggest is $H'(H(x)^d)$. To compute it, the client computes $H(x)$, obtains an RSA blind signature on it from the bank, and applies H' to the result.

1. c obtains blinded signature bx_r^d either from B or as protocol input.
2. c sends $bH(x)x_r^d$ to B.
3. B sends $H(H(x)x_r^d)$ to c.
4. c outputs $H(xH(H(x)x_r^d)$.

Figure 8.2: PRF Protocol: c obtains $f_r(x)$ from B

a relay. The PRF used by relay r is

$$f_r(x) = H(x(H(H(x)x_r^d))), \tag{8.1}$$

where, as described above, $x_r \in \mathbf{Z}_M^*$ is publicly known.

The PRF Protocol for client c to obtain $f_r(x)$ from the bank B is given in Figure 8.2. We leave the option to obtain a blind signature in Step 1 as an input to the protocol to enable a batch-mode execution that will be used in the final purchase protocol. The client will be unable to guess outputs that he doesn't query with better than random chance because the relay signature x_r^d never appears in a message from the server that hasn't been blinded or hashed. Moreover, the protocol protects the privacy of the client's inputs (doing so is what prevents us from using a simpler PRF, such as $H(x_r^d x)$). In particular, the first unblinded input the bank sees has a factor $H(x)$ and thus appears random given that the bank doesn't know x. Including a factor of x before applying H in the last step prevents the bank itself from learning the final output.

Private Evaluation of Pseudorandom Permutations

We will now consider how to turn this into a permutation. Given $f : \{0,1\}^k \to \{0,1\}^k$, the *Feistel permutation* $D_f : \{0,1\}^{2k} \to \{0,1\}^{2k}$ on $x = x_1||x_0$, $x_1, x_0 \in \{0,1\}^k$, is defined as

$$D_f(x_1||x_0) = (x_0||x_1 \oplus f(x_0)). \tag{8.2}$$

This is invertible because x_0 is contained in the first k bits, and x_1 can be calculated as $f(x_0) \oplus (x_1 \oplus f(x_0))$. Luby and Rackoff showed that applying the Feistel permutation four times with four pseudorandom functions yields a pseudorandom permutation. We use this idea, but, in our setting, we will disallow winners at a relay that result in PRF

1. c randomly chooses a and sends $a^e x_r$ to B.
2. B randomly chooses b and sends bax_r^d to c.
3. c uses the PRF protocol with input bx_r^d to obtain $f_r(y_1)$ and sets $y_2 = f_r(y_1) \oplus y_0$.
4. c uses the PRF protocol with input bx_r^d to obtain $f_r(y_2)$ and sets $y_3 = f_r(y_2) \oplus y_1$.
5. c uses the PRF protocol with input bx_r^d to obtain $f_r(y_3)$ and sets $y_4 = f_r(y_3) \oplus y_2$.
6. c uses the PRF protocol with input bx_r^d to obtain $f_r(y_4)$ and sets $y_5 = f_r(y_4) \oplus y_3$.
7. c outputs $y_5||y_4$.

Figure 8.3: PRP Protocol: c obtains $g_r^{-1}(y)$ from B

inputs that have been used before. Thus, we can use the Luby-Rackoff construction with the single pseudorandom function f_r. Thus we obtain a PRP for relay r of

$$g_r(x) = D_{f_r}\left(D_{f_r}\left(D_{f_r}\left(D_{f_r}\left(x\right)\right)\right)\right). \tag{8.3}$$

The PRP in Equation 8.3 is used by the relay to determine if a given ticket is a winner. To purchase a winner, a client will actually choose a permutation output from the winning range and apply the inverse permutation to obtain the ticket number. Let such an output be $y = y_1||y_0$, $y_0, y_1 \in \{0,1\}^{\lambda/2}$. The PRP Protocol for a client c to obtain $g_r^{-1}(y)$ from the bank B is given in Figure 8.3. Observe that this protocol gives the client quite a bit more information about the function g^{-1} than a simple oracle query would. In fact, it reveals enough information for the client to determine values of g which he has not obtained via the protocol. However, this is only possible for a negligible quantity of inputs and we will show that relays can limit him to obtaining guaranteed priority only as many times as he has paid for winners (see Section 8.4).

Protocol to Purchase Winners

The Winner Purchase Protocol run by client c to purchase a winner for relay r from the bank B is given in Figure 8.4. We emphasize that the bank will only participate in step 3 of the Winner Purchase Protocol if c presents a valid coin. This protocol is run entirely over an anonymous onion-routing connection made by c to B. To prevent

1. c pays B a digital coin.
2. c randomly chooses $y \in [1, p2^\lambda]$.
3. c uses the PRP protocol to obtain $g_r^{-1}(y)$.

Figure 8.4: Winner Purchase Protocol: c purchases a winner for r from B

the bank from learning when encrypted circuits were made, clients should buy enough winners at a time to construct γ prioritized circuits, should maintain a reserve of enough winners to construct $\gamma/2$ prioritized circuits, and after depleting their reserve below this amount should wait a minimum time selected uniformly at random from $[0, \omega]$ before purchasing more winners.

We set γ to balance privacy with respect to the bank with the flexibility of the recommended buying strategy. Increasing γ increases the amount of time between batches of purchases observed by the bank and thus the number of other users' prioritized circuits that hide the buyer's circuits. On the other hand, decreasing γ decreases the number of coins a relay should have stockpiled before purchasing winners. We will ask a relay to run for at least 3 hours before purchasing winners. For a relay providing the current median bandwidth in Tor of 100 KiB/s [30] and with Tor's path length of $\ell = 3$, after 3 hours the relay obtains about 79 coins. Each prioritized circuits costs ℓ coins, and so we set $\gamma = 26$.

We set ω to maximize the number of prioritized connections that could have triggered a purchase without emptying the reserve of winners. Thus we set ω to the point at which a client's reserve of $\gamma/2$ winners would become empty had he been making only prioritized connections. For clients that make new circuits at rate r this happens at $\omega = \gamma/(2r)$. In Tor, $r \approx 1$, and so we have $\omega = 13$ minutes.

8.3.4 Circuit Setup

LIRA slightly modifies the onion-routing circuit-creation protocol (cf. Section 8.2.1) to accommodate prioritization. Clients use a new TICKET cell type to send a lottery number to each relay on a circuit and attempt to obtain priority. A TICKET cell has the structure [TICKET, *number*], where the *number* field contains a value in $[1, 2^\lambda]$. This cell is sent to a relay from the client over the circuit and is thus onion-encrypted. In addition, relays on a circuit signal prioritization status to one other. These messages

1. c runs onion-routing circuit-creation protocol.
2. For each relay r in the circuit, c sends a TICKET cell with either a purchased winner w_r or random guess $x_r \in [1, 2^\lambda]$.
3. For every β bytes that pass over the circuit, c repeats Step 2 with new winners or guesses.

Figure 8.5: Circuit Setup Protocol for client c

are sent directly over an encrypted pairwise connection (*e.g.* over the persistent TLS connections that Tor maintains between pairs of relays) with an identifier indicating to which circuit they pertain.

The Circuit Setup Protocol at the client is described in Figure 8.5. It simply adds periodic lottery-ticket messages to the standard circuit-creation protocol in onion routing. Note that the ticket messages are sent onion-encrypted over the circuit and thus can only be read by the recipient.

Relays determine priority for their circuits using the Circuit Priority Protocol described in Figure 8.6. For each position in a circuit, a relay maintains priority values for itself, for the preceding segment of the circuit, for the succeeding segment of the circuit, and for the entire circuit. The relay also maintains a counter for the number of priority bytes that are left under the current prioritization. Observe that the protocol involves explicit signaling along the circuit to synchronize the priority status of the relays. Thus, for the circuit depicted at the bottom of Figure 8.1(c), even though the middle relay has received a winner, it will mark the circuit as DEAD after it receives a DEAD relay priority from either the guard or exit relay.

Also observe that, during the validation of a ticket number, the PRF inputs used during the four applications of the Feistel permutation (Eq. 8.3) are stored. If a PRF input used during ticket validation has been seen before, then that ticket is considered a loser. This prevents a client from reusing old winners and from using PRF outputs obtained from the bank to construct multiple winners. Finally, note that once a losing ticket has been observed, the priority status is DEAD and no further priority is possible on the circuit.

The value of β, the number of bytes for which a winner provides priority, provides

```
Upon receiving data cell
  If priority bytes counter is greater than zero
    Reduce priority bytes counter by cell size
  If priority bytes counter is zero
    and self priority status is not DEAD
    Set self priority status to EXPIRED
    Set circuit priority status to EXPIRED.
Upon receiving priority status from adjacent relay
  Store status and send to other adjacent relay
  If all stored relay priorities are TRUE
    Set circuit priority TRUE
  If any stored relay priority is DEAD
    Set circuit priority DEAD
Upon receiving TICKET cell with value $x$
  Compute $g_r(x)$, storing intermediate PRF inputs
  If $g_r(x) \in [1, p2^\lambda]$
    and intermediate PRF inputs previously unseen
    and no stored priority status is DEAD
    Set self priority status to TRUE
    Increment priority bytes counter by $\beta$
    Set circuit priority TRUE
  Else set self and circuit priority statuses to DEAD
  Send self priority status to adjacent relays
```

Figure 8.6: Circuit Priority Protocol for relay r

a tradeoff between user anonymity and the incentive to run a relay. A small β makes it unlikely that a *guesser*, that is, a user who does not buy winners, will maintain priority over the life of a typical circuit. This reduces the anonymity of a *buyer*, who we wish to allow to obtain priority for an entire connection. Setting a large β, assuming the price of a winning ticket increases proportionally, causes buyers to either use a circuit for a long time, reducing their anonymity by increasing the linkability of their connections, or to lose many of the bytes for which priority has been purchased, decreasing the incentive to earn it. Also, a large β increases the granularity of buying winners, and so more e-cash must be earned before it can be used, again decreasing the incentive to earn e-cash.

Given these considerations, we use a value of β that is greater than the length of a typical connection. As discovered by McCoy et al. [28], over 90% of connections over Tor are HTTP connections. Ramachandran[80] shows data from billions of web pages showing that the mean size, including all embedded content, is 320 KiB. Cheng et al. [143] show that the mean YouTube file size is 8.4 MiB. To enable most web connections as well as such popular activities as viewing videos, we use $\beta = 10$MiB.

8.3.5 Circuit Scheduling

To improve the quality of service of qualifying traffic—cells on circuits for which valid winners have been provided—we incorporate ideas from the differentiated services architecture (DiffServ) [86]. More specifically, LIRA schedules circuits using *Proportional Throughput Differentiation* as described and evaluated in Chapter 5 (Section 5.3). Scheduling in this model allows us to configure the performance payoff associated with running a relay, or correctly guessing a winning ticket. Note that LIRA solves the scheduling classification problem by differentiating circuits that submitted winners from those that did not.

8.4 Analysis

8.4.1 Efficiency

LIRA preserves all the communication functionality of onion routing while providing both anonymity and efficiency. This section will consider how LIRA affects the overall computational and communication costs of the network.

Overhead

Clients may purchase a winner from the bank for each relay in a circuit to receive $\beta = 10$ MiB of prioritized traffic. Purchasing these winners involves ℓ RSA encryptions for the client portions of blind signatures and 2ℓ hashes. This cost is on the order of the cost for building a circuit, which is continuously incurred throughout a Tor client session. Clients that do not purchase winners incur no extra computational cost by using LIRA.

Our goal is to keep relay CPU costs low because, according to the Tor developers, the high-bandwidth Tor relays are CPU-bound. LIRA introduces some overhead for relays with ticket verification, *i.e.*, checking whether or not tickets are winners. This process involves evaluating the PRP in Equation 8.3 for every $\beta = 10$ MiB of transferred data. Each evaluation involves 12 hash computations, as well as a smaller number of multiplications and XORs. The DiffServ scheduler has been shown to be efficient [87] since each scheduling decision must only compute one priority for each class.

The bank is involved in distributing e-cash to relays and selling winning tickets. To generate e-cash, the bank creates a coin for every $\alpha/\ell = 10(\ell+1)/\ell$ MiB sent by a given relay. Creating a coin involves a single blind signature, and these coins are given to the relays using a simple two-message protocol. To sell a winner, the bank must verify a coin by verifying a signature, provide a blind signature, and then participate in the batch PRF Protocol four times, each of which involves one hash.

We now consider the bank costs if the entire network is transferring b MiB/s and the fraction of coins that end up being used by the relays that earn them is f. Let $\rho = \ell b/(10(\ell + 1))$ be the rate at which coins are generated in this network. The rate of costly cryptographic operations and communicated messages for each bank service are outlined in Table 8.1. It shows that the rate of the cryptographic operations is just a fraction of the total rate of traffic on the network. Table 8.1 also shows that the communication costs at the bank, in terms of the number of messages, the size of a digital coin, and the size of a ticket number, are similarly small.

Table 8.1: Bank Costs

Service	Operations	Messages
Coin generation	ρ signatures	ρ coin-size sent ρ coin-size received
Selling winners	ρf verifications $4\rho f$ hashes ρf signatures	ρf coin-size sent $2\rho f$ coin-size received $4\rho f$ winner-size sent $4\rho f$ winner-size received

Current Costs in Tor

To get a more concrete idea of what the costs at the bank might be in practice, we estimate what it would cost for a bank to serve the current Tor network. In Tor, $b = 1700$ and $\ell = 3$. The most costly operation by one to two orders of magnitude is signature generation. In the above setting, the rate of signature generation is $127.5 + 127.5f$ per second, where again $f \in [0, 1]$. OpenSSL benchmarks in Linux on an Intel Core2 Duo 2.67 GHz machine show that it is capable of creating 1705 1024-bit RSA signatures per second, thus a modest machine is easily capable of generating the required signatures.

We also estimate the communication costs at the bank in this setting by using a signature size of 1024 bits and a ticket-number size of $\lambda = 320$ bits. Then the bank sends at a total rate of $15.94+35.86f$ KiB/s and receives at a total rate of $15.94+51.80f$ KiB/s, easily manageable with a single consumer-grade network connection.

Comparison to BRAIDS

To further understand LIRA's efficiency, we compare it to the efficiency of BRAIDS (Chapter 7 [46]), the state-of-the-art Tor relay incentive design. The cost for each client in LIRA and BRAIDS are similar, however, only the clients that are purchasing guaranteed winners must pay this cost in LIRA as opposed to all clients that receive free tickets in BRAIDS. Further, a relay verifying winners in LIRA is at least an order of magnitude more efficient than a relay verifying tickets in BRAIDS: our OpenSSL benchmarks indicate LIRA's winner verification (12 SHA-1 hashes) takes roughly 36 microseconds, whereas a BRAIDS ticket verification takes roughly 1500 microseconds (see Table 7.1) on the same hardware.

The most important cryptographic cost is at the bank. LIRA's bank only needs to cryptographically pay the relays for some fraction of the total traffic due to its lottery design. On the other hand, the bank in BRAIDS pays for all traffic by distributing tickets to all clients. In addition, the number of ticket purchases is proportional to the amount of e-cash actually used by the relays to obtain prioritized service. If, as we expect, many relays altruistically provide more service than needed to support their own use, the system gains significant efficiency over distributing tickets to clients.

If we change the parameters of the BRAIDS scheme to more conservatively compare LIRA[3], we observe that BRAIDS requires at least 637.5 signatures per second. Even if $f = 1$ and relays spend all their credit, LIRA is more efficient. Moreover, BRAIDS requires a less computationally efficient partially-blind signature scheme. The signature-generation protocol in BRAIDS also has higher communication costs in both directions than RSA blind signatures, and thus is easily greater than LIRA's costs in both directions.

8.4.2 Anonymity

We are interested in the extent to which the use of LIRA affects the anonymity of onion routing. Onion routing security is well-explored in the literature [138, 144, 17, 19, 32], and its vulnerabilities generally exist after adding our incentive system. Therefore, our goal is to prevent the anonymity provided by onion routing from being significantly degraded. In this section we denote by $\mathsf{negl}(\lambda)$ a function that is negligible in λ.[4]

Single Connection

Consider first a network adversary observing a single connection below the priority cutoff length of β. If the adversary is observing at a guard node, the user may be identified

[3]We suppose that BRAIDS creates tickets for only half of the network traffic, as it is designed to cover Web traffic (58% of Tor traffic [28]). Also, we allow each ticket to buy 10MiB of priority, as in LIRA, and we give relays a bytes sent/earned ratio of 1/4, also as in LIRA. We then have (1700*3)/(2*4*10)=63.75 tickets created per second, which yields 63.75*20/2 = 637.5 total signatures per second when ticket exchanges are included.

[4]A function f that is negligible in λ decreases faster with λ than any inverse polynomial. That is, $f(\lambda) = o(1/\lambda^k)$ for any k.

as running a relay if he is connecting to the guard from it. However, the multiple connections case shows that this would be quickly learned by the guard anyway. Thus it is a good idea for the user to use his own relays as guards.

Assuming the adversary is not observing at a guard node, from the adversary's perspective, LIRA simply adds TICKET cells and status signaling messages between pairs of relays. Users only directly affect the TICKET cells. Guessers and buyers generate these cells according to different distributions, potentially leading to some deanonymization. The difference lies in the probability that the TICKET cells contain a winner or not. Therefore, we need only consider an adversary's observations about whether or not the user inserts winning tickets into each of the relays on a circuit.

Suppose that a user does provide winners for an entire circuit. This happens with probability 1 for buyers and p^ℓ for guessers. The adversary can easily learn that submitted tickets were winners if he controls one of the circuit's relays. He may also learn this by observing the speed of traffic to and from a destination under his observation. Over time, the adversary can learn the distribution of traffic speeds for prioritized and unprioritized traffic and use the separation between these (as demonstrated in Section 8.5) to infer the priority status of a given observed connection. Assume that buyers consist of the m relays and guessers consist of the other users of the network at a given time, and that each user is *a priori* equally-likely to create a circuit. Then the probability that the source of a connection is a given relay, based only on its circuit prioritization status, is $1/(m+(n-m)p^\ell)$, and the probability that it is a given non-relay is $p^\ell/(m+(n-m)p^\ell)$.

Now suppose that a user's tickets are not winners for the entire circuit. This happens with probability 0 for buyers and $1-p^\ell$ for guessers. As before, the adversary can determine this in several ways. Buyers never fail to provide a winner, and so the adversary can infer that the source is a guesser. Given $n-m$ guessers, the probability that the source is a given one is $1/(n-m)$. (Of course buyers could intentionally fail to submit winning tickets at some relays periodically to complicate this analysis. We do not evaluate such possibility in this chapter.)

We are most interested in the case that there are relatively few buyers, as that would be true currently if LIRA were deployed in Tor, and we would expect it to remain so as long as the cost of running a relay is high relative to the benefit of anonymity for most users. In this case, the probability that a given buyer is the source of a single short prioritized connection, based only on its circuit prioritization status, is roughly $1/(m + np^{\ell})$. With $p = n^{-1/(2\ell)}$, this becomes $1/(m + \sqrt{n})$. Thus, we can see that uncertainty over the source increases with the total number of users n, a desirable property of onion routing that we want to preserve. With few buyers, the probability that a given guesser is the source of a single short unprioritized connection, based only on the circuit's prioritization status, is roughly $1/n$, which is the best possible.

Of course, an adversary need not only take into account the prioritization status of a relay for purposes of deanonymization. Indeed, as discussed, all attacks on onion routing itself maybe be used in addition to the information provided by LIRA. However, the action of the incentive system is independent of the underlying onion routing protocol, and therefore the effect on deanonymization is simply to weight the distribution an adversary would otherwise infer. For example, suppose that, excluding the observations from the incentive system, the adversary can infer that the source is a given user with probability p_1. If that user has probability p_2 of achieving the priority observed, then, including those observations, the probability of the user becomes (proportional to) $p_1 p_2$. One consequence of this as that LIRA increases the posterior probability of a buyer by at most $1/p^{\ell}$.

Multiple Connections

Circuits on which more than β bytes are sent include multiple TICKET cells from users. The above analysis applies to any one priority status, but taken together they degrade the anonymity of the user to the point that they are essentially identified as either a buyer or a guesser. Suppose a user's circuit transfers more than $(k-1)\beta$ bytes. This will happen when a single connection exceeds that amount. It can also happen if the total volume of multiple connections sent over the same circuit exceed that amount.[5] In this situation the user updates his priority status k times. The probability that a guesser maintains priority through all the updates is $p^{k\ell}$. Therefore, such a circuit created by a buyer quickly identifies him as a buyer, and the probability that it is a given buyer conditional only on the priorities observed is $1/(m + (n - m)p^{k\ell})$. Guessers are always

[5]The amount of traffic sent over a circuit depends on the relative rates at which circuits and connection are created and destroyed. Tor only puts new connections on circuits that have been used for less than ten minutes, with a preference among used circuits for the youngest.

identified as guessers when the tickets they submit fail to be winners, and this happens with an increasing likelihood of $1 - p^{k\ell}$.

Users making many circuits over time face the possibility of a similar decrease in anonymity. If an adversary can observe the priority status of k of a user's connection and link them together as belonging to the same (unknown) user, the resulting anonymity is just as if the user updated the priority status of a given circuit k times. Some ways the adversary may be able to link connections include controlling a destination at which users are active over long-lived sessions, controlling some exit nodes and linking together connection by related activities, and controlling some middle nodes and observing connections coming from the same guard nodes. The adversary is also be able to link connections at a guard node because they come from the source directly. A user running a relay may hope to hide that fact from a given guard by connecting to the anonymity network from a different location and buying tokens from a guard anonymously through a different guard. However, if he consistently buys priority, his guards will quickly determine that.

Hiding over the long term the fact that better service is being purchased seems to be a fundamental issue that any scheme will suffer from to some degree. In BRAIDS, for example, normal users receive fewer coins than relays, and so they can only be confused with relays if they save up many coins before buying, and thus few of them can buy at any one time. On the other hand, allowing users to purchase service without running a relay, which we ignored in our analysis due to uncertainty, has the potential to attract many more users than those that run relays. The Torservers.net project [135] demonstrates that many prefer donating money to running relays. (Note that this also presents a mechanism whereby purchasing priority can indirectly add commensurate capacity to the network if all proceeds of such sales are directed into the purchase of more capacity, such as Torservers.net does.) Further, the widespread use of VPNs for Internet security and blocking resistance indicate a willingness to pay for privacy.

Bank Privacy

We assume that the bank is semi-honest and only observes messages sent to it. The bank only observes the amount of e-cash earned by relays, when cash is transferred among users, and the purchase of winners. Clearly, then, the bank doesn't learn anything about

1. B sends challenge relays $r_0 \neq r_1$ to C.
2. C chooses a random bit $z \in \{0,1\}$.
3. C executes the Winner Purchase Protocol with B for relay r_z.
4. B outputs a guess z' for the value of z.
5. The experiment value is 1 if $z' = z$, 0 if not.

Figure 8.7: WPP-REL-IND experiment between B and C

the destinations of connection through the anonymity network, and therefore users have relationship anonymity with respect to the bank. However, LIRA protects user privacy even further.

First, all purchases at the bank are made using anonymous connections and anonymous coins. Therefore the bank doesn't learn who is spending e-cash and buying service.

Second, clients batch and randomly time their purchase of ticket winners to hide when prioritized circuits are made from the bank. Clients should purchase $\gamma\ell$ winning tickets at a time. If a relay prioritizes all his circuits and makes them at Tor's rate $r \approx 1$ per minute, he purchases winners every $\gamma = 26$ minutes. Moreover, the time of the purchase triggered by a prioritization that reduces a client's reserve of winners below the $\gamma/2$ threshold is hidden from a bank within a period of $\omega = 13$ minutes. To get an idea of how many other prioritizations occur during these time periods, consider $n = 10000$ users making circuits at rate r, each gaining priority with probability $p^\ell = 1/\sqrt{n}$. Then during a 26 minute period between purchases there are an expected $\gamma n p^\ell r = 2600$ prioritized circuits from other users and during a 13 minute period there are an expected 1300 such circuits.

Third, the Winner Purchase Protocol hides from the bank the relay identity and the ticket number of a purchased winner. The indistinguishability experiment WPP-REL-IND between the bank B and a challenger C shown in Figure 8.7 tests how well the bank can determine the relay of a purchase. Theorem 1 shows that the observations a bank makes during a purchase for given relay are indistinguishable from the observations made during a purchase for a different relay.

Theorem 1 *In the Random Oracle Model, $Pr[\mathsf{WPP\text{-}REL\text{-}IND} = 1] \leq 1/2 + \mathsf{negl}(\lambda)$.*

Proof 1 *Model H as a Random Oracle. Right before the bank makes a guess, he has* *made a polynomial number of queries to H. During the execution of the Winner Purchase Protocol for relay r_z, the bank provides a blind signature and participates in four executions of the batch PRF Protocol. In providing the blind signature, the bank receives the value $a^e x_{r_z}$ from the client. This value is random from the bank's perspective because a is chosen randomly by the client. The bank also receives $bH(y_i)x^d_{r_z}$, $1 \leq i \leq 4$. The bank knows b, $x^d_{r_0}$, and $x^d_{r_1}$, and so he can tell if any $H(y_i)x^d_{r_z}$ is equal to $H(x)x^d_{r_{z''}}$ for some x that he has queried to H and $z'' \in \{0,1\}$. Let Query be the event that the bank has queried H on some such x. If Query occurs, we will assume that $z' = z$, and the experiment value is 1. If Query doesn't occur, then each $H(y_i)$ must be on some y_i not queried by B to H. Thus both sets of potential values of the $H(y_i)$, one for $z = 0$ and one for $z = 1$, are equally likely from the bank's perspective. In this case $z' = z$ with probability 1/2.*

Thus we simply need to show that $Pr[\mathsf{Query}]$ is negligible. The initial input y_1 is distributed randomly from B's perspective. Therefore the probability that B has queried H for y_1 is negligible. y_2 is the result of $H(y_1 H(H(y_1)x^d_{r_z}))$. y_1 is random to B, and thus the probability that B has queried H on this value is negligible. Assuming this, y_2 is random in B's view. The probability that y_3 and y_4 are non-random to B is negligible for similar reasons. Because with high probability each y_i is random to in B view, $Pr[\mathsf{Query}] = \mathsf{negl}(\lambda)$.

We can define an experiment similar to WPP-REL-IND to test how well the bank can guess the ticket number of a purchase. It can be shown that the bank succeeds in that experiment with at most a negligible amount over a random guess as well. This exposition is omitted for space reasons.

8.4.3 Incentives

LIRA is designed to create an incentive for users to run relays or otherwise contribute to the system. We explore in Section 8.5 the extent to which it successfully provides better service to users that receive priority. Here we consider if users must, as we intend, earn e-cash in order to increase the amount of priority they can obtain. Again we denote by $\mathsf{negl}(\lambda)$ a function that is negligible in λ.

We first note that the possibility of cheating the system is an important consideration but one less important than performance and preserving anonymity. Regardless of whether or not a user can deviate from the protocol to obtain more priority, the anonymity and performance properties of the system still hold. Thus system operators could experiment with the use of LIRA without compromising the properties the network already provides. Furthermore, the amount of cheating that will occur in practice in a network protocol is unclear. BitTorrent, for example, is susceptible to cheating [145, 146], but it tends to perform well in practice. Somewhat low barriers to cheating may well be sufficient to induce most participants to comply.

LIRA is designed to force users to pay in order to obtain a winner with probability greater than p. It achieves this by using an e-cash scheme and a novel cryptographic lottery. In the e-cash scheme, users must present valid digital coins to participate in the Winner Purchase Protocol (Figure 8.4). The e-cash scheme prevents coin forgery as well as double spending.

The winners themselves are obtained by participating in the PRP Protocol (Figure 8.3). This protocol allows users to observe much more about g_r than just the output, and thus it is not true that g_r in fact appears as a PRP. However, the intermediate PRF outputs that the users observe are only allowed by a relay to appear in one winner. If another ticket is submitted with a previously seen PRF value, the relay will treat it as a loser. Thus these intermediate values are of no use in producing more winners than were paid for, and on inputs with unseen intermediate PRF values, the lottery permutation g_r does indeed appear random.

We formalize this property in the security experiment PRP between an adversary A and a challenger C shown in Figure 8.8. We will show that A succeeds in this experiment with at most a negligible probability greater than a random chance. But first, we show that the PRF Protocol actually provides the PRF properties. The PRF experiment shown in Figure 8.9 tests whether, after executing the PRF Protocol some arbitrary number of times t, an adversary A can distinguish $f_{r_c}(x)$ from a random value for more than t inputs x, where r_c is some challenge relay. Lemma 1 shows that the adversary succeeds in this experiment with a probability that is at most a negligible amount over random chance.

1. A outputs relays $r = \{r_1, \ldots, r_k\}$ and a challenge relay $r_c \notin r$.
2. C outputs random values $\{x_{r_1}, \ldots, x_{r_k}, x_{r_c}\}$ in $\mathbf{Z}_M^*$ and signatures $\{x_{r_1}^d, \ldots, x_{r_k}^d\}$.
3. C executes the PRP Protocol with A as many times t as requested.
4. A outputs $x = \{x_1 \ldots, x_t\}$ and $\{y_1, \ldots, y_t\}$ and an input x_c for g_{r_c}.
5. C randomly chooses $z \in \{0, 1\}$. If $z = 0$, C sends $g_{r_c}(x_c)$, else C sends a random y.
6. A outputs a guess z' for z.
7. If $\forall_i y_i = f_{r_c}(x_i)$, no intermediate PRF input for $g_{r_c}(x_c)$ appears in x, and $z' = z$, the experiment value is 1; else it is 0.

Figure 8.8: PRP experiment between A and C

1. C generates RSA parameters (M, e, d) and outputs (M, e).
2. A outputs relays $r = \{r_1, \ldots, r_k\}$ and a challenge relay $r_c \notin r$.
4. C outputs random values $\{x_{r_1}, \ldots, x_{r_k}, x_{r_c}\}$ in $\mathbf{Z}_M^*$ and signatures $\{x_{r_1}^d, \ldots, x_{r_k}^d\}$.
5. C executes the PRF Protocol with A some t times.
6. A outputs $x = \{x_1, \ldots, x_t\}$, $\{y_1, \ldots, y_t\}$, $x_c \notin x$.
7. C randomly chooses $z \in \{0, 1\}$.
8. C outputs $f_{r_c}(x_c)$ if $z = 0$ and else a random y.
9. A outputs a guess z'. The experiment value is 1 if $z' = z$, and $\forall_i y_i = f_{r_c}(x_i)$. Otherwise it is 0.

Figure 8.9: PRF experiment between A and C

Lemma 1 *In the Random Oracle Model and under the RSA assumption,* $Pr[\mathsf{PRF} = 1] \leq 1/2 + \mathsf{negl}(\lambda)$.

Proof 2 *We can reduce winning this game to solving the RSA problem. We construct a simulator* S *for the* PRF *challenger* C *and random oracle* H*.* S *implements* H *internally.* S *implements parameter selection by relaying RSA parameters from the RSA challenger to* PRF *adversary* A*.* S *randomly selects the relay signatures* $x_{r_i}^d$*, computes the relay values* x_{r_i} *from them, and randomly selects* x_{r_c}*.* S *executes the challenger side of the PRF Protocol by storing the response values* $w_i = H(x_i)$ *output by* A *and using random values* v_i *for each* $H(H(x_i)x_{r_c}^d)$*.*

Step 1 of the PRF Protocol is random, and so the distribution of these values given A*'s view is accurately produced by* S*. The value for* $H(H(x_i)x_{r_c}^d)$ *is indeed random in* A*'s view unless the response from* A *in Step 2 of the PRF Protocol is such that* $x_{r_c}^d w_i$ *is equal to some value queried of* H *by* A*.* S *can check for this possibility by dividing all queries for* H *by each new* w_i *received from* A*, raising it to the power* e*, and comparing to* x_{r_c}*. If any verification succeeds,* S *submits that value to the RSA challenger. Under the RSA assumption, the probability that this happens is negligible. Assuming this doesn't happen,* C *randomly chooses a bit* z *and outputs a random* y*. If the outputs* x_i *and* y_i *of* A *are such that* $H(x_i) = w_i$ *and* $y_i = H(v_i x_i)$*, then* S *has not "programmed"* H *with a value for* $H(x_c)x_{r_c}^d$ *(i.e. choosing a value without knowing the input).* S *again checks if* $H(x_c)x_{r_c}^d$ *has been queried by dividing all queries by* $H(x_c)$ *and raising to the* e*, sending to the RSA challenger if so. Thus this happens with negligible probability. Assuming it hasn't,* $f_{r_c}(x_c)$ *is random from* A*'s perspective, and the random* y *from* C *has the correct distribution.* z *is random and independent of* y*, and thus* $z' = z$ *with probability 1/2.*

Therefore, overall C *presents the correct view to* A *except with negligible probability, and thus* $Pr[\mathsf{PRF} = 1] \leq 1/2 + \mathsf{negl}(\lambda)$*.*

Using Lemma 1, we can now show that the adversary cannot find inputs to the PRP g_r that look non-random and don't collide with previous input-output pairs obtained via the PRP Protocol:

Theorem 2 *In the Random Oracle Model,* $Pr[\mathsf{PRP} = 1] \leq 1/2 + \mathsf{negl}(\lambda)$*.*

Proof 3 *The proof for the pseudorandomness of the Luby Rackoff construction [141] requires different PRFs for each Feistel permutation only to guarantee that they are* *independently pseudorandom with respect to the previous Feistel rounds. However, our experiment prevents the adversary from succeeding if the inputs to f_{r_c} occur more than once during an evaluation of the permutation g_{r_c}. Assuming this doesn't occur, by Lemma 1, the output of f_{r_c} in each Feistel round is independent of the previous rounds.*

While this theorem shows that users can only themselves produce unused winners with probability p, a user may try to game the network by creating circuits and determining their priority. If he attempts to do so without colluding with any relays, he must determine the priority of the circuit from its performance alone. Suppose that doing so requires that he send or receive at least c cells on a circuit. Then, to obtain a circuit with priority, the user must transfer an expected c/p total cells. If the cost of this is comparable to the amount of traffic a relay needs to transfer in order to earn enough coins to build a circuit, a rational user might choose to take that more-reliable option.

A user might also run or collude with a relay in order to obtain priority without paying. A relay on a circuit is able to determine from the messages between adjacent relays on a circuit its priority status. Therefore, a user could collude with a guard node to create and destroy circuits until one with priority is obtained. Similarly, the user could collude with a middle or exit node, although given that the user in this case is presumably known to the colluding relay, it would seem only to improve performance without decreasing anonymity to directly connect to that relay. A simple remedy for this attack that renders it equivalent to testing and creating circuits is to defer the activation of priority on a circuit until some number c of cells have passed in either direction. The initial traffic on a circuit is fast even without priority due to EWMA scheduling, and so the performance impact should be minimal, although we have not implemented it in our experiments. An additional possible attack is for a colluding relay in the middle of the circuit to lie about the priority status of either side in order to get partial priority of a circuit. However, we again observe that it would seem to make little sense for a user to use a colluding relay as a middle node rather than connect directly. Also, for all attacks that involve a relay, the costs associated with running a relay are already being paid, and it would have to be the case that the cost of simply adding capacity is more than the cost of running a cheating scheme.

Finally, we observe that similar opportunities for cheating exist in other recent incentive schemes for anonymous communication. The Tortoise scheme of Moore et al. [94] allows users to create many circuits to avoid throttling, similar to the multiple-circuits attacks in LIRA. Also, the BRAIDS design from Chapter 7 is susceptible to malicious guards as well, in that guards can easily steal the winning tickets intended for their users. Thus while LIRA does not eliminate cheating, it does offer a substantially new balance among competing priorities.

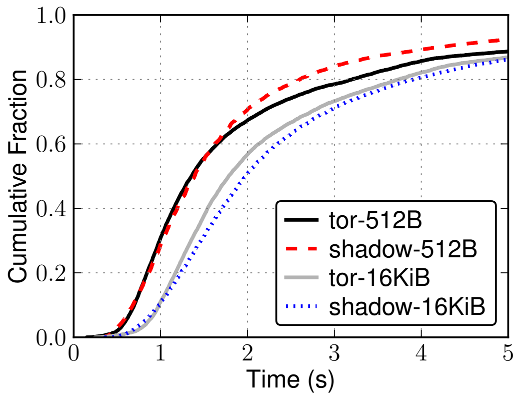
8.5 Experiments

We simulate LIRA in an effort to understand the performance benefits possible when running our incentive scheme. Our experiments are done using Shadow (Chapter 3 [62]), a scalable, high-fidelity network simulator that is capable of running real Tor binaries as plug-ins (using the available Shadow plug-in called Scallion [51]). Shadow allows us to create a private Tor network on a single machine and avoid privacy risks associated with live network experiments. Shadow experiments are completely controllable and repeatable, and are faithful to Tor's protocols since Shadow runs the real Tor software. In this section, we describe our configured experimental network environment, quantify its consistency with public Tor network performance, and explore how LIRA affects performance and improves incentives for a variety of users. Note that all of the experiments described in this section are repeated ten times to diminish random experimental variances, and each uses Tor software version `0.2.3.13-alpha`.

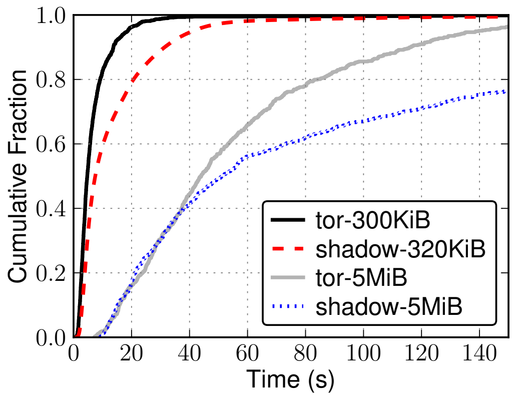
8.5.1 Network Model

Shadow requires a complete-graph network topology that includes properties such as upstream and downstream bandwidths, latency, jitter, and packet loss. As network modeling is itself a challenging research problem, we rely on previous Tor network modeling contributions from Chapter 4 [147]. Their work considers every element of the Internet and the Tor network itself that must be modeled to run accurate Tor experiments in Shadow. Their model is built using real Internet measurements from GeoIP [60], iPlane [73, 78], and Net Index [75], and is validated with multiple experimentation platforms and data from the live Tor network itself [30].

We now give an overview of the Tor network model used in our experiments and discuss how it was modified from the original, the full details of which are presented in



(a) IM and P2P Clients



(b) Web and Bulk Clients

Figure 8.10: Shadow and public Tor performance are reasonably consistent for various transfer sizes.

Chapter 4. Our private Tor network consists of 50 generic HTTP servers, 50 Tor relays, 500 Tor clients. Of the 50 Tor relays, there is 1 directory authority, 20 exit relays, and 29 non-exit relays.

Although the original model configured 475 web and 25 bulk clients, a wider range of client applications would provide a more realistic traffic distribution. Therefore, we slightly modify the clients to better approximate Tor's protocol distribution as described in [28] and [29]. We configure 10 instant messaging clients (im), 465 web HTTP clients (web), 20 bulk HTTP clients (bulk), and 5 peer-to-peer clients (p2p). The im clients download 1 KiB files, pausing for one to five seconds after finishing one download and before starting the next. The web clients download 320 KiB files and pause for 1 to 20 seconds. The bulk clients continuously download 5 MiB files without pausing. All of the im, web, and bulk clients choose a random HTTP server for each download. The p2p clients form a "swarm" around a single 700 MiB file that is managed by a p2p authority. Each pair of p2p nodes connect and continuously exchange 16 KiB blocks of the file without pausing. Payload download times are measured as an indication of network performance.

8.5.2 Model and Simulation Accuracy

As we modified the model as originally described and validated in Chapter 4, we re-evaluate the consistency of Shadow's results with live network data. To determine how well our modeled network approximates client performance in the public Tor network, we compare download times in a vanilla Tor experiment with measurements of Tor collected by the TorPerf measurement system [83]. TorPerf downloads 50 KiB, 1 MiB, and 5 MiB files through the Tor network to monitor performance, and records various download times. Because our clients download differently sized files than TorPerf, we compare the time to receive the last byte of each of our experimental downloads with the time to receive the closest byte that is reported by TorPerf. As shown in Figure 8.10, Shadow does a reasonable job of characterizing the expected performance of the public Tor network. Performance for im and p2p clients are consistent with TorPerf measurements (Figure 8.10(a)), as are web and bulk downloads below approximately the fiftieth percentile (Figure 8.10(b)).

The difference in performance in the upper half of the distributions is possibly due to Tor's scheduling policy [33], in which circuit priority decreases as its throughput increases. TorPerf will have higher expected priority than clients in our experiments since TorPerf downloads once per circuit whereas our clients download multiple times per circuit. Note that we were unable to confirm this suspicion beyond reasonable doubt due to a lack of experimental single file circuit downloads.

8.5.3 LIRA Prototype

We implement a research prototype of LIRA as described in Section 8.3 by directly modifying the Tor source code. To understand how to attribute changes in performance, we run separate experiments using the default EWMA circuit scheduling algorithm (vanilla Tor), our new Proportional Throughput Differentiation scheduler (diffserv) based on work by Dovrolis *et al.* [87] (see Section 8.3.5), and various LIRA configurations (lira).

Class Differentiation

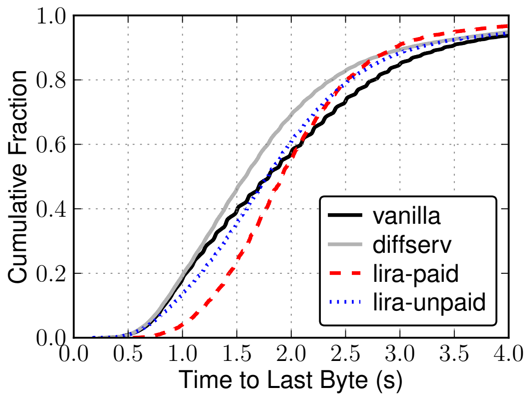
We configure our new prototype scheduler with "paid" and "unpaid" classes c_1 and c_2, and differentiation parameters $p_1 = 1.0$ and $p_2 = 10.0$. Priorities are weighted by taking fraction $f = 0.875$ of the head-of-queue circuit EWMA, and 0.125 of the long-term class average EWMA. The EWMA throughput algorithms in both classes are configured with a 30 second half-life, which is also the default in our vanilla experiment and in public Tor. In our diffserv experiment, we isolate the new scheduler from the LIRA prototype: all clients are categorized in the unpaid class and there is no ticket guessing or buying. For each relay in our lira experiments, there is a corresponding client who uses that relay's winners to receive priority for all of its downloads. Of these 50 "paid" clients, we configure 1 im client, 47 web clients, 1 bulk client, and 1 p2p client. The remaining clients are "unpaid" and will only receive a prioritized circuit by correctly guessing with probability $p = 0.01$. Each prioritized circuit may be used for $\beta = 10$ MiB of data transfer, after which new guesses are submitted.

As shown by the cumulative distributions of download times in Figure 8.11, the new scheduler appears to give slightly preferential service to low throughput im clients and slightly worse to high throughput p2p clients. The scheduler tends to perform slightly worse than Tor's default scheduler, possibly because our prototype implementation has not been optimized. Our diffserv experiment provides a base upon which LIRA may be compared. The fundamental mechanism provided by the scheduler that is used to create performance incentives is tunable class differentiation. Figure 8.11 shows the scheduler's ability in this regard, as paid downloads are clearly differentiated from unpaid downloads. Note that the loss in performance for paid im downloads in Figure 8.11(a) is an artifact of the small sample – a single im client on a high latency link due to unfavorable placement in the network topology.

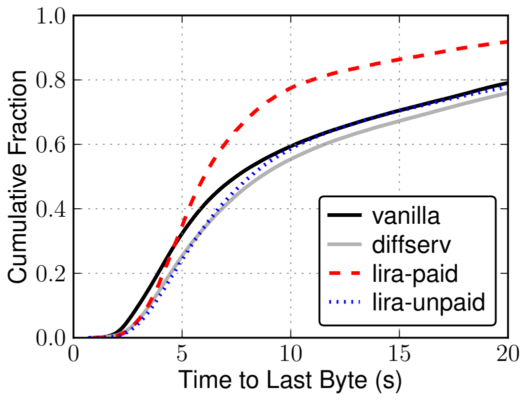
New Relay Capacity

Figure 8.11 shows performance in a network where only the existing Tor relays receive priority. We now explore a situation where several existing clients begin routing traffic for Tor. We consider networks where 5% and 15% of the existing client base[6] begin running a relay, adding a total of 25 and 75 relays and newly-paid clients to the existing sets of 50. Rationally, each new relay severely rate-limits its contribution so as to earn only enough winning tickets to support the expected throughput requirements of its

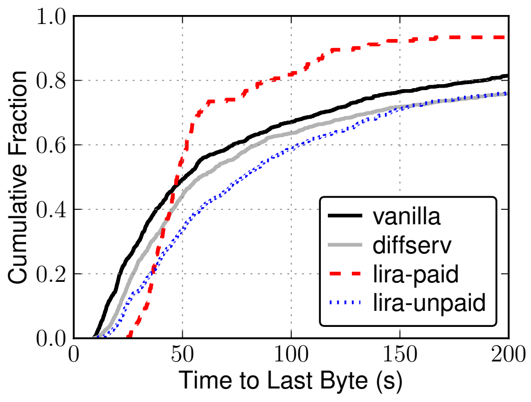
[6]Of the 5% of clients that begin running relays, we select 1 im, 23 web, 1 bulk, and 1 p2p. Of the 15%, we select 2 im, 69 web, 2 bulk, and 2 p2p.



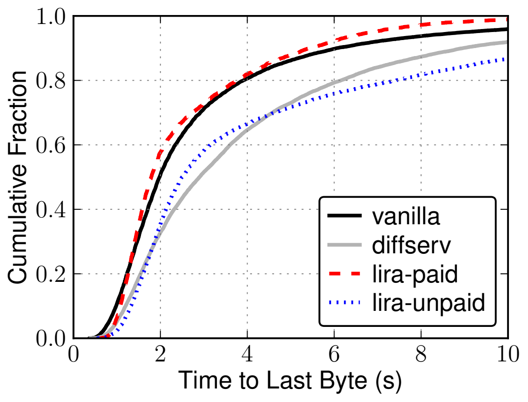
(a) 1 KiB IM Clients



(b) 320 KiB Web Clients



(c) 5 MiB Bulk Clients



(d) 16 KiB P2P Clients

Figure 8.11: Client download time distribution in vanilla Tor, when using our proposed scheduler, and after adding LIRA's design modifications. LIRA adequately differentiates performance for paying clients without additional capacity.

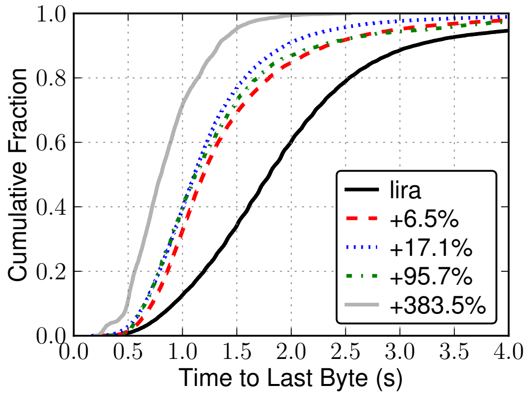
client as computed from our vanilla experiment. This is a conservative estimate. Each client contributes four times its expected client throughput since LIRA will prioritize 1/4 of a relay's contributed bytes when $\ell = 3$. Therefore, rate-limits are set to 20 KiB/s for those running im clients, 80 KiB/s for those running web clients, 340 KiB/s for those running bulk clients, and 128 KiB/s for those running p2p clients. Note that 1/3 of the added relays are exit relays, roughly the same proportion as in the current Tor network.

The rate-limiting outlined above results in 6.5% and 17.1% total additional network capacity, and represents a slightly pessimistic approximation of expected client contributions. To understand both extremes of the range of possible user behaviors, we configure other networks where the same 15% of new relays chosen above do not rate-limit their contributions. In the non-rate-limited networks, new relay bandwidth is sampled from the Net Index distribution [75] and results in a 95.7% and 383.5% increase in network capacity. Figure 8.12 shows the result of additional capacity on client performance. As expected and not surprisingly, the added capacity results in a net increase in overall performance over LIRA without new relays, even under our rate-limiting scenarios. The net benefit to the network increases for all clients types, and more dramatically as more capacity is added. Our results confirm that LIRA (and the proportional throughput scheduler) enables performance incentives for contributors.

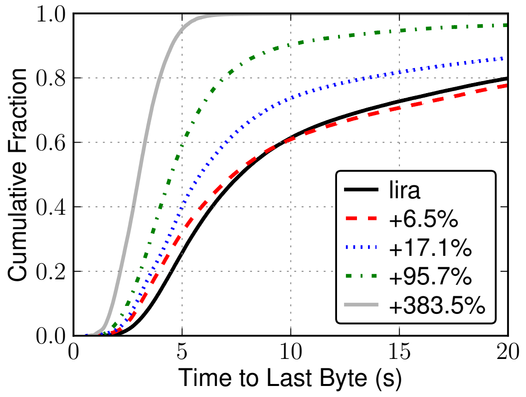
8.6 Summary

The Tor network suffers from performance problems partially caused by a lack of relays willing to altruistically volunteer bandwidth. This chapter presented LIRA, a novel incentive scheme that increases performance for those who contribute to the network by running a relay. We have shown that clients who choose to run relays enjoy faster downloads than those who don't, due to a novel ticket lottery design and a scheduler that differentiates service for winning tickets.

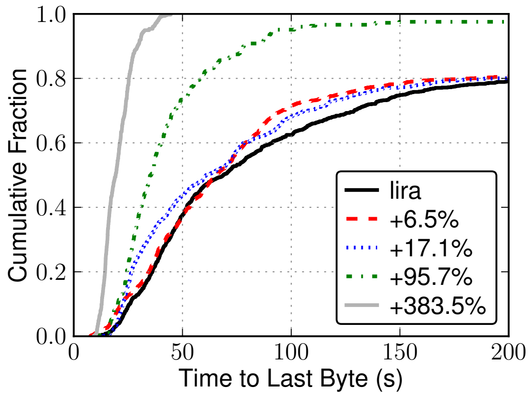
LIRA provides a higher degree of anonymity than previous proposals while eliminating the need for clients to contact the bank since, with tunable probability, clients can randomly self-produce winning tickets.



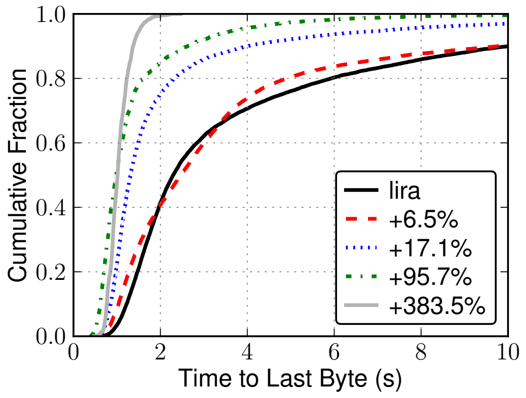
(a) 1 KiB IM Clients



(b) 320 KiB Web Clients



(c) 5 MiB Bulk Clients



(d) 16 KiB P2P Clients

Figure 8.12: Client download time distribution as clients begin to run relays. Adding additional capacity as shown results in a net increase in performance considering both paid and unpaid clients.

Chapter 9

Related Work

9.1 Experimentation

This section reviews several experimentation techniques that have been used to test Tor's performance and resistance to various attacks. A test environment that accurately reflects Tor's behavior is crucial to produce meaningful results. We now briefly explore experimentation techniques chosen by researchers to evaluate Tor proposals. We note that: Kiddle [148] provides a comprehensive analysis and discussion of system simulation and emulation techniques; Naicken *et al.* [149, 150] provide details on several generic simulators; and Bauer *et al.* [65] provide an in-depth survey of experimental approaches historically used in Tor-related research.

9.1.1 Simulation

Simulation typically involves creating abstract models of system processes and running multiple nodes in a single unified framework. Experiment management is simplified since there are many fewer simulation host machines (typically one) than simulated nodes. By abstracting system processes, simulators can run much more efficiently and are not required to run in real time. However, the abstraction process has the potential to reduce accuracy since the simulator may not encompass complex procedures that may in fact be important to system interaction. Although simulation platforms exist [151, 152, 153, 154], they are not capable of running unmodified versions of the Tor software.

Simulation has often been employed for Tor research, but simulators tend to be written for a specific problem and may be difficult to apply to a generic context: Murdoch and Watson explore Tor path selection strategies and algorithms [49], O'Gorman and Blott simulate packet counting and stream correlation attacks [50], and Ngan *et al.* study the effects of their gold-star priority scheme on Tor performance [47]. These simulators have either become unmaintained or are not publicly available, making published results challenging to validate.

9.1.2 Emulation

A competing and fundamentally different experimentation approach involves emulation. An emulator "tricks" an application or operating system that it is running on its own physical machine, when in fact it is virtualized in software. Emulators require a large amount of overhead to ensure the emulated software runs in real time while providing the virtualization layers needed to emulate an entire system. Therefore, emulation is potentially more accurate than simulation, but *much less scalable*: emulators typically run hundreds of nodes while simulators run thousands.

Due to intensive resource requirements, emulation platforms often utilize a large testbed of geographically distributed physical hardware. Examples of whole-system emulation testbeds include PlanetLab [48] and DETER [64]. Both of these frameworks only supply a few hundred nodes to a user. Several Tor studies have utilized the PlanetLab and DETER testbeds for experimenting with traffic analysis attacks [41, 155, 19], attacks on Tor bridges [156], and relay circuit scheduling [33]. Due to resource consumption and co-location of nodes on each physical machine, results on these testbeds often suffer from a reduced and false sense of accuracy. Further, distributed (e.g., PlanetLab) experiments are challenging to manage and control while results are difficult to recreate.

A Tor emulation testbed has recently been simultaneously and independently proposed by Bauer *et al.* [65] based on the ModelNet emulation platform [68]. The emulation testbed, called ExperimenTor, works by configuring multiple host machines with new operating system installations. Some of these host machines run a version of ModelNet link emulators while the remaining machines run Tor and other application instances. Tor nodes are given IP addresses from separate virtual interfaces to allow multiple nodes per machine while sending all traffic over the ModelNet hosts to emulate configured network properties.

LIRA has several advantages over ExperimenTor despite having similar goals and motivations. First, LIRA is more *usable* than ExperimenTor, which requires multiple physical machines, kernel modifications, and complex configuration. LIRA can be run as a stand-alone user application without root privileges and requires little configuration, leading to an extremely small barrier to entry and improving accessibility to students, developers, and researchers around the world. Second, LIRA is more *efficient* and *scalable* than ExperimenTor. LIRA implements a discrete-event simulator which allows full utilization of computational resources while eliminating the requirement of running in real time: experiments may run either faster or slower than real time without affecting accuracy. Conversely, ExperimenTor suffers from both CPU and bandwidth bottlenecks: the CPUs on the machines running the ExperimenTor testbed must run at far less than

100 percent utilization and the aggregate traffic load from all application instances must not exceed the capacity of the physical network connecting the host machines. Both requirements must be met to ensure the emulated applications do not lag, since lag would skew and invalidate results obtained in an experiment. LIRA also minimizes the memory overhead of running multiple applications on a single machine with its "state swapping" approach to memory management whereas ExperimenTor duplicates entire copies of the application in memory. Finally, LIRA allows for a richer *customization* of the experimental process, e.g. adversarial entities could easily be added to links between nodes to allow monitoring of network level traffic. Similar customizations would be difficult to add to an ExperimenTor testbed.

9.2 Performance Tuning

We now enumerate the wide range of recent work on improving Tor's performance.

9.2.1 Quality of Service

Networks often want to provide a certain quality of service (QoS) to their subscribers. There are two main approaches to QoS: Integrated Services (IntServ) and Differentiated Services (DiffServ).

In the IntServ [103, 104] model, applications request resources from the network using the resource reservation protocol [105]. Since the network must maintain the expected quality for its current commitments, it must ensure the load of the network remains below a certain level. Therefore, new requests may be denied if the network is unable to provide the resources requested. This approach does not work well in an anonymity network like Tor since clients would be able to request unbounded resources without accountability while the network would be unable to fulfill most requests due to congestion and bottlenecks.

In the DiffServ [86] model, applications notify the network of the desired service type by setting bits in the IP header. Routers then tailor performance toward an expected notion of fairness (e.g. max-min fairness [85, 98] or proportional fairness [99, 34, 110]). Leaking this type of information about a client's traffic flows is a significant risk to privacy. Please note that techniques for providing differentiated service without such risk were explored in Chapter 5 (Section 5.3).

9.2.2 Scheduling

Alternative Tor circuit scheduling approaches have recently gained interest. Tang and Goldberg [33] suggest each relay track the number of packets it schedules for each circuit. After a configurable time-period, packet counts are exponentially decayed so that data sent more recently has a greater influence on the packet count. For each scheduling decision, the relay flushes the circuit with the lowest cell count, favoring circuits that have not sent much data recently while preventing bursty traffic from significantly affecting scheduling priorities. In Chapter 5, we investigate new schedulers based on the proportional differentiation model [110] and differentiatable service classes. Relays track quality metrics for each service class and prioritize scheduling so that relative quality is proportional to configurable differentiation parameters, but the schedulers require a mechanism for differentiating traffic into classes. Finally, Tor's round-robin TCP read/write schedulers have recently been noted as a source of unfairness for relays that have an unbalanced number of circuits per TCP connection [39]. Tschorsch and Scheuermann suggest that a round-robin scheduler could approximate a max-min algorithm [85] by choosing among all circuits rather than all TCP connections. More work is required to determine the suitability of this approach in the live Tor network.

Scheduling algorithms, such as fair queuing [157] and round robin [84, 85], affect the order in which packets are sent out of a given node, but generally do not change the total number of packets being sent. Therefore, unless the sending rate is explicitly reduced, the network will still contain similar load regardless of the relative priority of individual packets. As explained in Section 6.1 and Section 6.2, scheduling does not directly reduce network congestion, but may cooperate with other bandwidth management techniques to achieve the desired performance characteristics of traffic classes.

9.2.3 Congestion

Improving performance and reducing congestion has been studied by taking an in-depth look at Tor's circuit and stream windows [40]. AlSabah *et al.* experiment with dynamically adjusting window sizes and find that smaller window sizes effectively reduce queuing delays, but also decrease bandwidth utilization and therefore hurt overall download performance. As a result, they implement and test an algorithm from ATM networks called the N23 scheme, a link-by-link flow control algorithm. Their adaptive N23 algorithm propagates information about the available queue space to the next upstream router while dynamically adjusting the maximum circuit queue size based on outgoing cell buffer delays, leading to a quicker reaction to congestion. Their experiments indicate slightly improved response and download times for 300 KiB files.

9.2.4 Relay Selection

Snader and Borisov [43] suggest an algorithm where relays opportunistically measure their peers' performance, allowing clients to use empirical aggregations to select relays for their circuits. A user-tunable mechanism for selecting relays is built into the algorithm: clients may adjust how often the fast relays get chosen, trading off anonymity and performance while not significantly reducing either. It was shown that this approach increases accuracy of available bandwidth estimates and reduces reaction time to changes in network load while decreasing vulnerabilities to low-resource routing attacks. Wang *et al.* [97] propose a congestion-aware path selection algorithm where clients choose paths based on information gathered during opportunistic and active measurements of relays. Clients use latency as an indication of congestion, and reject congested relays when building circuits. Improvements were realized for a single client, but its unclear how the new strategy would affect the network if used by all clients.

9.2.5 Transport

Tor's performance has also been analyzed at the socket level, resulting in suggestions for a UDP-based mechanism for data delivery [44] or using a user-level TCP stack over a DTLS tunnel [35]. While Tor currently multiplexes all circuits over a single kernel TCP stream to control information leakage, the TCP-over-DTLS approach suggests separate user TCP streams for each circuit and sends all TCP streams between two relays over a single kernel DTLS-secured [158] UDP socket. As a result, a circuit's TCP window is not unfairly reduced when other high-bandwidth circuits cause queuing delays or dropped packets.

9.3 Incentives

We now discuss incentives in Tor and other distributed systems.

9.3.1 Tor Incentives

A recognition that Tor is limited by its bandwidth resources has resulted in several proposals for developing performance incentives for volunteering bandwidth as a Tor relay. New relays would provide additional resources and improve network performance. Incentive designs for Tor include PAR [113], a scheme where relays accept real monetary payments from clients in return for routing service. PAR separates payments into anonymous coins paid by clients to guard relays, and more efficient identity-bound coins paid to the remaining relays. PAR and a similar micropayment scheme called XPAY [114] require an online bank to participate in the routing protocol to verify that the coins have not been double-spent. The need for the bank to frequently verify coins introduces a fundamental design problem – a trade-off between double spending detection and anonymity: the bank may use coins to launch an intersection attack. Neither LIRA (Chapter 8) nor BRAIDS (Chapter 7) suffer from the fundamental trade-off between double-spending detection and accountability that plagues PAR and XPAY, wherein anonymity inherently decreases as the ability to detect cheaters improves.

An alternative to using e-cash or other payment-based cryptographic mechanisms to provide incentives, Ngan *et al.* propose a lighter-weight scheme in which the fastest 7/8 relays are marked with a "gold star" in the public Tor directory based on measurements by the directory servers [47]. These relays are given priority as they build circuits through other gold-star relays, and enjoy improved performance because only fast relays receive gold stars. Unfortunately, relay anonymity is reduced because the set of potential initiators of a prioritized circuit (the gold-star relays) is much smaller than that of an unprioritized circuit (any active client). Not only is the anonymity set of relays is significantly reduced since gold star relays can be distinguished from regular relays, but also the changing membership of the gold star set leads to an *intersection attack* [17, 19, 115, 20]. Both LIRA and BRAIDS manages the small anonymity set problem by allowing every user to receive priority.

Tortoise [94] is another lightweight alternative incentive scheme. Moore *et al.* suggest in Tortoise a universal rate limit of Tor clients and an exemption from such throttling for relays marked as `stable` and `fast` in the consensus. Not only must Tortoise's throttling configurations must be monitored as network load changes but also Tortoise provides less anonymity than the gold star scheme. In both systems, the timing of relays' priority status appearing in the consensus leaks information that enables an intersection attack over time. However, the intersection attack is improved for the adversary in Tortoise since gold star nodes retain their gold stars for several months after dropping from the consensus, whereas Tortoise only unthrottles nodes that are in the current consensus. Tortoise clients may also multiplex traffic over multiple guards to evade throttling, thereby weakening the incentives provided by the system.

9.3.2 Incentives in Other Networks

Incentives have been previously proposed for several anonymous and peer-to-peer systems. Both Anonymizer.com [159] and the Freedom network [160] introduced commercial anonymity systems based on collecting payments for service. While the latter failed, the former is still in operation and provides a one-hop anonymous proxy based system for paying clients.

Franz *et al.* [161] introduce an incentive technique for mix-networks that divides electronic payments for each mix, but it is inefficient since each hop requires communication between the mix and mix provider. Figueiredo *et al.* [162] also introduce an electronic mix-net anonymity system, but it lacks accountability and robustness. Reiter *et al.* [112] build upon coin ripping [111] to develop a strict fair exchange protocol for mix-nets. However, they require each message recipient to participate in the protocol which does not align with Tor's desire to support arbitrary destinations.

Golle *et al.* [163] discuss incentives for sharing in peer-to-peer networks using a game theoretic model. They propose a micro-payment or "points" system, where uploads are rewarded and downloads are penalized, and show that equilibrium is reached by balancing uploads and downloads. BitTorrent [164] uses "tit-for-tat" mechanisms to trade pieces of large files among peers. Peers upload and download pieces cooperatively and preferentially from other peers in an attempt to maximize local download efficiency. The Scrivener system [165] uses a credit/debit approach to maintain local histories of other nodes' cooperative behavior which is used to enforce fair bandwidth sharing. Similarly, reputation systems [166, 167, 168, 169] use interaction histories to develop trust levels for other peers which aids in future decisions to cooperate while incentivizing trustworthy behavior. Reputation systems are incompatible with Tor since not all relays are able to bind an interaction to a client.

There has also been some work considering the behavior of participants in an anonymous communication network from an economic perspective. Acquisti et al. [45] describe the costs and benefits of anonymity-network users, identify the challenges in designing systems that cope with selfishness, and present some possibilities for solving those challenges. They argue a usage fee as an economic incentive mechanism, however, cost is then a security objective since it affects the number of users and therefore the anonymity provided [170]. Humbert et al. [171] provide an analysis of using a scrip system to incentivize selfish agents in a cooperative privacy-enhancing system such as an anonymity network. They establish the existence of a Nash equilibrium, examine its social welfare, and show how to manage the supply of scrip. Future study of sociological behaviors in LIRA would be interesting should it be adopted.

Chapter 10

Conclusion and Future Work

After discussing the importance of safeguarding electronic communication, this dissertation explored an experimentation tool called Shadow and its methodologies for safely measuring modifications to Tor. We then presented design modifications to Tor that attempt to enhance performance for Tor's users while maintaining the level of privacy Tor currently provides. Our designs and evaluations focussed on resource utilization through a variety of scheduling mechanisms, reducing network load through several throttling techniques, and two system designs that create performance incentives for users to run relays in order to increase network capacity.

10.1 Future Work

We now enumerate several open questions raised in this dissertation and discuss directions for future research.

10.1.1 Network Experimentation and Modeling

There are several architectural modifications that can improve Shadow's run-time performance. The most significant improvement will enhance Shadow's ability to run in parallel environments, leading to faster experiments and better utilization of hardware resources. Shadow may be used to explore a wide range of problems in Tor, including alternative transport mechanisms, validation of previous work, and analysis of Tor attacks under various network configurations and client models.

There are several ways in which our Tor network model could be improved. First, we could increase the size of the network by improving the software support and acquiring the hardware resources necessary for handling larger networks in the available experimentation tools. Running at or near scale means we may reduce experimentation artifacts, such as those created because relay selection probabilities necessarily change when using only a subset of the existing relays. Larger networks will also provide a more realistic experimentation environment and more realistic results.

Second, our model may benefit from capacity and link characteristics gathered directly from Tor relays. This would give us precise statistics about the specific nodes we are modeling and reduce our reliance on external sources of more generic information for links between relays. One possible approach to capacity measurement involves using packet trains [79], but more work is needed to determine the efficacy of such techniques in the context of the Tor network. At the same time, many research questions may require a more detailed topology structure than that modeled in this dissertation. Higher fidelity of the underlying network topology may be possible by combining data from both the iPlane[78] and the CAIDA[74] data sets.

Third, determining a better client model would further increase confidence in experimental results. Producing a more robust client model will likely require the development of algorithms for collecting client statistics in a way that mitigates privacy risks. While this is challenging since client behaviors are dynamic and hard to capture in a representative fashion, it would allow us to increase faithfulness to the live Tor network and its users. Finally, modeling malicious adversaries and their behaviors may be of specific interest to future research that analyzes the security of Tor or its algorithms.

10.1.2 Resource Management

A deeper analysis of the intricacies of Tor clients and relays would improve our understanding of how various loads and configurations affect our scheduling and throttling algorithms. Our current scheduling and throttling algorithms may be modified to improve performance by replacing our heuristics with a more accurate classification of bulk traffic, considering alternative strategies for distinguishing web from bulk connections. Also of interest is an analysis of scheduling and throttling in the context of congestion and flow control to determine the interrelation and effects the algorithms have on each other. Finally, a deeper understanding of our algorithms and their effects on client performance would be possible through analysis on the live Tor network.

10.1.3 Incentives to Contribute

There are several issues in our BRAIDS design that need further investigation. Our uniform ticket validity intervals introduce a trade-off between fast ticket turn-around times and offline relays losing their tickets. The trade-off is due to our imposed ticket tax which is required to bound the bandwidth load on the bank. Tickets you earn will not be usable for 2 days in our current design, and you will lose half of your tickets if you are offline for a day and unable to exchange them.

BRAIDS would benefit from a more robust ticket distribution scheme, since an adversary controlling a fraction of IP addresses in Tor can steal a similar fraction of tickets in each spending interval, and malicious agents receive a greater reward for stealing tickets than behaving honestly. Further, auditing ticket agents would allow us to detect cheaters.

Future work may also explore client strategies to enhance our model of client spending habits and improve our anonymity analysis. Users must be aware of their prioritized spending habits since spending a large number of tickets (more than a few hundred MB) will reduce their anonymity and lead to the risk of intersection attacks as in the gold star scheme. Finally, distributing the bank's functionality among users, or a small set of trusted nodes, will reduce bandwidth and CPU limitations, drastically improving anonymity.

Among numerous possibilities for improving LIRA is developing a better understanding of the economics of anonymous incentives and how rational users might be expected to behave in LIRA or a similar design. Also useful would be a modified incentive structure that provides non-linear payoff for contributed capacity and higher payoff for more desirable relays such as bridges, exits, and those in more diverse geographic locations. A distributed bank that functions securely within Tor's trust model would improve scalability. Finally, better defenses against strategies for attempting to cheat the system and improved protection against long-term anonymity problems associated with linking paid high-throughput users would not only benefit LIRA, but any system attempting to provide anonymity-protected incentives.

10.2 Final Remarks

Shadow is open-source software that we feel is invaluable for understanding and evaluating distributed systems like Tor. Our tools and techniques improve our ability to reason about Tor's design, and this dissertation has provided the foundations for evaluating and designing new algorithms and protocols that further enhance Tor's performance and anonymity. Although there is still much work to be done, we are optimistic that our work has improved Tor's position as an elite, practical, and usable tool for maintaining online privacy for an increasing fraction of the population.

References

[1] EFF Sues AT&T to Stop Illegal Surveillance. https://www.eff.org/press/archives/2006/01/31. Accessed July, 2012.

[2] The New York Times Reminds Us the NSA Still Warrantlessly Wiretaps Americans, and Congress Has the Power to Stop It. https://www.eff.org/deeplinks/2012/08/ny-times-reminds-us-nsa-still-warrantlessly-wiretapping-americans-and-congress-has. Accessed August, 2012.

[3] Packet Forgery By ISPs: A Report on the Comcast Affair. https://www.eff.org/wp/packet-forgery-isps-report-comcast-affair. Accessed July, 2012.

[4] And the Privacy Invasion Award Goes To . https://www.eff.org/deeplinks/2012/05/and-privacy-invasion-award-goes-to. Accessed August, 2012.

[5] AOL Releases Search Logs of 657,427 Users. http://slashdot.org/story/06/08/07/2022244/aol-releases-search-logs-of-657427-users. Accessed July, 2012.

[6] Facebook 'Deceived' Users By Sharing Private Information. http://www.huffingtonpost.co.uk/2011/12/01/facebook-guilty-of-privacy-breach_n_1122749.html. Accessed July, 2012.

[7] http://donttrack.us/. Accessed July, 2012.

[8] 2011 in Review: Internet Freedom in the Wake of the Arab Spring. https://www.eff.org/deeplinks/2011/12/2011-review-internet-freedom-wake-arab-spring. Accessed August, 2012.

[9] Users of Tor. https://www.torproject.org/about/torusers.html.en.

[10] David Goldschlag, Michael Reed, and Paul Syverson. Hiding routing information. In *Information Hiding: First International Workshop*, pages 137–150. Springer-Verlag, LNCS 1174, 1996.

[11] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: The second-generation onion router. In *Proceedings of the 13th USENIX Security Symposium*, 2004.

[12] The Tor Project. https://www.torproject.org/.

[13] Matthew Wright, Micah Adler, Brian Neil Levine, and Clay Shields. Defending Anonymous Communication Against Passive Logging Attacks. In *Proceedings of the 2003 IEEE Symposium on Security and Privacy*, pages 28–43, May 2003.

[14] Lasse Øverlier and Paul Syverson. Locating Hidden Servers. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*. IEEE CS, May 2006.

[15] Matthew Wright, Micah Adler, Brian Neil Levine, and Clay Shields. An Analysis of the Degradation of Anonymous Protocols. In *Proceedings of the Network and Distributed Security Symposium - NDSS '02*. IEEE, February 2002.

[16] Adam Back, Ulf Moller, and Anton Stiglic. Traffic analysis attacks and trade-offs in anonymity providing systems. In *Proceedings of Information Hiding Workshop (IH'01)*, pages 245–257, 2001.

[17] Nathan Evans, Roger Dingledine, and Christian Grothoff. A practical congestion attack on Tor using long paths. In *Proceedings of the 18th USENIX Security Symposium*, pages 33–50, 2009.

[18] Andrew Hintz. Fingerprinting websites using traffic analysis. In *Proceedings of Privacy Enhancing Technologies Workshop (PET'02)*, pages 171–178, 2002.

[19] Nicholas Hopper, Eugene Vasserman, and Eric Chan-Tin. How much anonymity does network latency leak? *ACM Transactions on Information and System Security (TISSEC'10)*, 13(2):1–28, 2010.

[20] Steven Murdoch and George Danezis. Low-cost traffic analysis of Tor. In *IEEE Symposium on Security and Privacy*, pages 183–195, 2005.

[21] Jean-Francois Raymond. Traffic analysis: Protocols, attacks, design issues, and open problems. In *Designing Privacy Enhancing Technologies*, pages 10–29, 2001.

[22] Andrei Serjantov and Peter Sewell. Passive attack analysis for connection-based anonymity systems. *Computer Security–ESORICS*, pages 116–131, 2003.

[23] Prateek Mittal, Ahmed Khurshid, Joshua Juen, Matthew Caesar, and Nikita Borisov. Stealthy Traffic Analysis of Low-Latency Anonymous Communication Using Throughput Fingerprinting. In *Proceedings of the 18th ACM conference on Computer and Communications Security*, October 2011.

[24] Roger Dingledine and Steven Murdoch. Performance improvements on tor or, why tor is slow and what were going to do about it. *Online: http://www. torproject. org/press/presskit/2009-03-11-performance. pdf*, 2009.

[25] Roger Dingledine and Nick Mathewson. Anonymity Loves Company: Usability and the Network Effect. In *Proceedings of the Fifth Workshop on the Economics of Information Security (WEIS 2006), Cambridge, UK, June*, 2006.

[26] Paul Syverson. Why I'm not an entropist. In *Seventeenth International Workshop on Security Protocols*. Springer-Verlag, LNCS, 2009. Forthcoming.

[27] Karsten Loesing. Measuring the Tor network: Evaluation of client requests to directories. Technical report, Tor Project, 2009.

[28] Damon Mccoy, Kevin Bauer, Dirk Grunwald, Tadayoshi Kohno, and Douglas Sicker. Shining light in dark places: Understanding the Tor network. In *Proceedings of the 8th International Symposium on Privacy Enhancing Technologies (PETS'08)*, pages 63–76, 2008.

[29] Abdelberi Chaabane, Pere Manils, and Mohamed Ali Kaafar. Digging into anonymous traffic: A deep analysis of the tor anonymizing network. In *Network and System Security (NSS), 2010 4th International Conference on*, pages 167–174. IEEE, 2010.

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

[31] Roger Dingledine. Iran Blocks Tor. https://blog.torproject.org/blog/iran-blocks-tor-tor-releases-same-day-fix.

[32] Nikita Borisov, George Danezis, Prateek Mittal, and Parisa Tabriz. Denial of service or denial of security? How attacks on reliability can compromise anonymity. In *Proceedings of CCS 2007*, October 2007.

[33] Can Tang and Ian Goldberg. An improved algorithm for Tor circuit scheduling. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, pages 329–339, 2010.

[34] Constantinos Dovrolis and Parameswaran Ramanathann. A case for relative differentiated services and the proportional differentiation model. *Network, IEEE*, 13(5):26–34, 1999.

[35] Joel Reardon and Ian Goldberg. Improving Tor using a TCP-over-DTLS tunnel. In *Proceedings of the 18th USENIX Security Symposium*, 2009.

[36] The Libevent event notification library, version 2.0. http://monkey.org/~provos/libevent/.

[37] Florian Tschorsch and Bjrn Scheuermann. Refill Intervals. https://gitweb.torproject.org/torspec.git/blob/HEAD:/proposals/183-refillintervals.txt.

[38] Research problem: adaptive throttling of Tor clients by entry guards. https://blog.torproject.org/blog/research-problem-adaptive-throttling-tor-clients-entry-guards.

[39] Florian Tschorsch and Bjrn Scheuermann. Tor is unfair–and what to do about it, 2011.

[40] Mashael AlSabah, Kevin Bauer, Ian Goldberg, Dirk Grunwald, Damon McCoy, Stefan Savage, and Geoffrey Voelker. DefenestraTor: Throwing out Windows in Tor. In *Proceedings of the 11th International Symposium on Privacy Enhancing Technologies (PETS'11)*, 2011.

[41] Kevin Bauer, Damon McCoy, Dirk Grunwald, Tadayoshi Kohno, and Douglas Sicker. Low-resource routing attacks against Tor. In *Proceedings of the 6th ACM Workshop on Privacy in the Electronic Society (WPES'07)*, pages 11–20, 2007.

[42] Lasse Overlier and Paul Syverson. Locating hidden servers. In *IEEE Symposium on Security and Privacy*, 2006.

[43] Robin Snader and Nikita Borisov. A tune-up for Tor: Improving security and performance in the Tor network. In *Proceedings of the 16th Network and Distributed Security Symposium (NDSS'08)*, 2008.

[44] Camilo Viecco. UDP-OR: A fair onion transport design. In *Proceedings of Hot Topics in Privacy Enhancing Technologies (HOTPETS'08)*, 2008.

[45] Alessandro Acquisti, Roger Dingledine, and Paul Syverson. On the economics of anonymity. In *Proceedings of the 7th International Conference on Financial Cryptography (FC'03)*, 2003.

[46] Rob Jansen, Nicholas Hopper, and Yongdae Kim. Recruiting new Tor relays with BRAIDS. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS'10)*, pages 319–328, 2010.

[47] Tsuen-Wan "Johnny" Ngan, Roger Dingledine, and Dan S. Wallach. Building incentives into Tor. In *The Proceedings of Financial Cryptography (FC'10)*, 2010.

[48] Brent Chun, David Culler, Timothy Roscoe, Andy Bavier, Larry Peterson, Mike Wawrzoniak, and Mic Bowman. Planetlab: an overlay testbed for broad-coverage services. *SIGCOMM Computer Communication Review*, 33:3–12, 2003.

[49] Steven Murdoch and Robert Watson. Metrics for security and performance in low-latency anonymity systems. In *Proceedings of the 8th International Symposium on Privacy Enhancing Technologies (PETS'08)*, pages 115–132, 2008.

[50] Gavin OGorman and Stephen Blott. Large scale simulation of Tor: Modelling a Global Passive Adversary. In *Proceedings of the 12th Conference on Advances in Computer Science – ASIAN*, pages 48–54, 2007.

[51] Shadow Development Repositories. http://github.com/shadow/.

[52] Shadow Homepage. https://shadow.cs.umn.edu/.

[53] Denis Foo Kune, Tyson Malchow, James Tyra, Nick Hopper, and Yongdae Kim. The Distributed Virtual Network for High Fidelity Large Scale Peer to Peer Network Simulation. Technical Report 10-029, University of Minnesota, 2010.

[54] The OpenSSL cryptographic library. http://www.openssl.org/.

[55] Jon Postel. User Datagram Protocol. RFC 768, http://www.ietf.org/rfc/rfc768.txt, August 1980.

[56] Jon Postel. Transmission Control Protocol. RFC 793, http://www.ietf.org/rfc/rfc793.txt, September 1981.

[57] The TorFlow measurement tools. https://gitweb.torproject.org/torflow.git/.

[58] Felix Hernandez-Campos, Kevin Jeffay, and F. Donelson Smith. Tracking the evolution of web traffic: 1995-2003. In *The 11th IEEE/ACM International Symposium on Modeling, Analysis, and Simulation of Computer Telecommunications Systems (MASCOTS'03)*, pages 16–25, 2003.

[59] The Iperf bandwidth measurement tool. http://iperf.sourceforge.net/.

[60] The MaxMind GeoIP Lite country database. http://www.maxmind.com/app/geolitecountry.

[61] Larry Peterson, Steve Muir, Timothy Roscoe, and Aaron Klingaman. PlanetLab Architecture: An Overview. Technical report, PlanetLab Consortium, 2006.

[62] Rob Jansen and Nicholas Hopper. Shadow: Running Tor in a Box for Accurate and Efficient Experimentation. In *Proceedings of the 19th Network and Distributed System Security Symposium*, 2012.

[63] Emulab Homepage. http://www.emulab.net.

[64] Terry Benzel, Robert Braden, Dongho Kim, Clifford Neuman, Anthony Joseph, Keith Sklower, Ron Ostrenga, and Stephen Schwab. Design, deployment, and use of the DETER testbed. In *Proceedings of the DETER Community Workshop on Cyber Security Experimentation and Test*, 2007.

[65] Kevin Bauer, Micah Sherr, Damon McCoy, and Dirk Grunwald. Experimentor: A testbed for safe and realistic tor experimentation. In *the 4th Workshop on Cyber Security Experimentation and Test (CSET'11)*, 2011.

[66] Stevens Le Blond, Pere Manils, Abdelberi Chaabane, Mohamed Ali Kaafar, Claude Castelluccia, Arnaud Legout, and Walid Dabbous. One Bad Apple Spoils the Bunch: Exploiting P2P Applications to Trace and Profile Tor Users. In *Proc. of the 4th Workshop on Large-Scale Exploits and Emergent Threats*, 2011.

[67] Karsten Loesing, Steven J. Murdoch, and Roger Dingledine. A Case Study on Measuring Statistical Data in the Tor Anonymity Network. In *Proc. of the Workshop on Ethics in Computer Security Research*, 2010.

[68] Amin Vahdat, Ken Yocum, Kevin Walsh, Priya Mahadevan, Dejan Kostić, Jeff Chase, and David Becker. Scalability and accuracy in a large-scale network emulator. *SIGOPS Operating Systems Review*, 36(SI):271–284, 2002.

[69] ExperimenTor Homepage. `http://crysp.uwaterloo.ca/software/exptor/`.

[70] Neil Spring, Ratul Mahajan, and David Wetherall. Measuring ISP Topologies with Rocketfuel. In *Proc. of the SIGCOMM Conference*, pages 133–145, 2002.

[71] Soon-Hyung Yook, Hawoong Jeong, and Albert-László Barabási. Modeling the Internet's Large-Scale Topology. *Proc. of the National Academy of Sciences*, 99(21):13382, 2002.

[72] Amogh Dhamdhere and Constantine Dovrolis. Twelve Years in the Evolution of the Internet Ecosystem. *IEEE/ACM Transactions on Networking*, 19(5):1420–1433, Sep 2011.

[73] Harsha Madhyastha, Tomas Isdal, Michael Piatek, Colin Dixon, Thomas Anderson, Arvind Krishnamurthy, and Arun Venkataramani. iPlane: An Information

Plane for Distributed Services. In *Proc. of the 4th Operating Systems Design and Implementation*, pages 367–380, 2006.

[74] CAIDA Data. http://www.caida.org/data.

[75] Net Index Data. http://www.netindex.com/source-data/.

[76] Bandwidth Speed Test. http://speedtest.net/.

[77] Ping Test. http://pingtest.net/.

[78] iPlane Data. http://iplane.cs.washington.edu/data.

[79] R. Jain and S. Routhier. Packet Trains–Measurements and a New Model for Computer Network Traffic. *IEEE Journal on Selected Areas in Communications*, 4(6):986 – 995, 1986.

[80] Sreeram Ramachandran. Web metrics: Size and number of resources. http://code.google.com/speed/articles/web-metrics.html, 2010.

[81] Sebastian Hahn and Karsten Loesing. Privacy-preserving Ways to Estimate the Number of Tor Users. Technical report, The Tor Project, November 2010. https://metrics. torproject. org/papers/countingusers-2010-11-30. pdf, 2010.

[82] Alexa The Web Information Company. Top 1 million sites. http://s3.amazonaws.com/alexa-static/top-1m.csv.zip. Retrieved January 31, 2012.

[83] TorPerf analysis tools. https://gitweb.torproject.org/torperf.git/.

[84] E.L. Hahne and R.G. Gallager. Round-robin Scheduling for Fair Flow Control in Data Communication Networks. *NASA STI/Recon Technical Report N*, 86:30047, 1986.

[85] E.L. Hahne. Round-robin scheduling for max-min fairness in data networks. *IEEE Journal on Selected Areas in Communications*, 9(7):1024–1039, 1991.

[86] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss. An Architecture for Differentiated Services, 1998.

[87] Constantinos Dovrolis, Dimitrios Stiliadis, and Parameswaran Ramanathan. Proportional differentiated services: Delay differentiation and packet scheduling. *IEEE/ACM Transactions on Networking (TON)*, 10(1):12–26, 2002.

[88] Including network statistics in extra-info documents. https://gitweb.torproject.org/torspec.git/blob_plain?f=proposals/166-statistics-extra-info-docs.txt.

[89] John D.C. Little and Stephen C. Graves. Little's Law. web.mit.edu/sgraves/www/papers/Little's%20Law-Published.pdf, 2008. Accessed July, 2011.

[90] Manuel Crotti, Maurizio Dusi, Francesco Gringoli, and Luca Salgarelli. Traffic Classification Through Simple Statistical Fingerprinting. *SIGCOMM Comput. Commun. Rev.*, 37:5–16, January 2007.

[91] E. Hjelmvik and W. John. Statistical Protocol Identification with SPID: Preliminary Results. In *Swedish National Computer Networking Workshop*, 2009.

[92] C. Kohnen, C. Uberall, F. Adamsky, V. Rakocevic, M. Rajarajan, and R. Jager. Enhancements to Statistical Protocol IDentification (SPID) for Self-Organised QoS in LANs. In *Computer Communications and Networks (ICCCN), 2010 Proceedings of 19th International Conference on*, pages 1–6. IEEE, 2010.

[93] Mashael AlSabah, Kevin Bauer, and Ian Goldberg. Enhancing Tor's Performance using Real-time Traffic Classification. In *Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS'12)*, 2012.

[94] W. Brad Moore, Chris Wacek, and Micah Sherr. Exploring the Potential Benefits of Expanded Rate Limiting in Tor: Slow and Steady Wins the Race With Tortoise. In *Proceedings of 2011 Annual Computer Security Applications Conference*, December 2011.

[95] Garrett Hardin. The tragedy of the commons. *Science*, 162(3859):1243–1248, December 1968.

[96] Fallon Chen and Mike Perry. Improving Tor Path Selection. https://gitweb.torproject.org/torspec.git/blob/HEAD:/proposals/151-path-selection-improvements.txt.

[97] T. Wang, K. Bauer, C. Forero, and I. Goldberg. Congestion-aware Path Selection for Tor. In *Proceedings of Financial Cryptography*, 2012.

[98] M. Katevenis. Fast switching and fair control of congested flow in broadband networks. *Selected Areas in Communications, IEEE Journal on*, 5(8):1315–1326, 1987.

[99] F.P. Kelly, A.K. Maulloo, and D.K.H. Tan. Rate control for communication networks: shadow prices, proportional fairness and stability. *Journal of the Operational Research society*, 49(3):237–252, 1998.

[100] Jonathan Turner. New directions in communications(or which way to the information age?). *IEEE communications Magazine*, 24(10):8–15, 1986.

[101] Research Code Repository. https://github.com/robgjansen/torclone.

[102] Rob Jansen, Paul Syverson, and Nicholas Hopper. Throttling Tor Bandwidth Parasites. Technical Report 11-019, University of Minnesota, 2011.

[103] B. Braden, D. Clark, and S. Shenker. Integrated Service in the Internet Architecture: an Overview, 1994.

[104] S. Shenker, C. Partridge, and R. Guerin. RFC 2212: Specification of Guaranteed Quality of Service, September 1997. Status: PROPOSED STANDARD.

[105] L. Zhang, S. Deering, D. Estrin, S. Shenker, and D. Zappala. Rsvp: A new resource reservation protocol. *Network, IEEE*, 7(5):8–18, 1993.

[106] Rob Jansen, Aaron Johnson, and Paul Syverson. Shadow: Running Tor in a Box for Accurate and Efficient Experimentation. In *Proceedings of the 20th Network and Distributed System Security Symposium*, 2013.

[107] Michael K. Reiter and Aviel D. Rubin. Crowds: Anonymity for web transactions. *ACM Transactions on Information and System Security*, 1(1):66–92, 1998.

[108] Tor partially blocked in China. Tor Project. https://blog.torproject.org/blog/tor-partially-blocked-china. October 2009.

[109] Constantinos Dovrolis and Parameswaran Ramanathann. Proportional differentiated services, part II: loss rate differentiation and packet dropping. In *IWQOS'00: Eighth International Workshop on Quality of Service*, pages 53–61, 2000.

[110] Constantinos Dovrolis, Dimitrios Stiliadis, and Parameswaran Ramanathan. Proportional differentiated services: delay differentiation and packet scheduling. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM'99)*, pages 109–120, 1999.

[111] M. Jakobsson. Ripping coins for a fair exchange. In *EUROCRYPT*, pages 220–230, 1995.

[112] M.K. Reiter, X.F. Wang, and M. Wright. Building reliable mix networks with fair exchange. In *ACNS'05: The Proceedings of the Third International Conference on Applied Cryptography and Network Security*, pages 378–392, 2005.

[113] Elli Androulaki, Mariana Raykova, Shreyas Srivatsan, Angelos Stavrou, and Steven M. Bellovin. PAR: Payment for anonymous routing. In *PETS '08: Proceedings of the 8th International Symposium on Privacy Enhancing Technologies*, pages 219–236, 2008.

[114] Yao Chen, Radu Sion, and Bogdan Carbunar. XPay: practical anonymous payments for Tor routing and other networked services. In *WPES '09: Proceedings of the 8th ACM Workshop on Privacy in the Electronic Society*, pages 41–50, 2009.

[115] Jon McLachlan and Nicholas Hopper. Don't clog the queue! Circuit clogging and mitigation in P2P anonymity schemes. In *FC'08: The Proceedings of the 12th International Conference on Financial Cryptography and Data Security*, pages 31–46, 2008.

[116] D. Chaum, A. Fiat, and M. Naor. Untraceable electronic cash. In *CRYPTO '88: Proceedings of Advances in Cryptology*, pages 319–327, 1990.

[117] Y. Tsiounis. Efficient electronic cash: new notions and techniques. *College of Computer Science*, 1997.

[118] Ronald L. Rivest and Adi Shamir. PayWord and MicroMint: Two simple micropayment schemes. In *Proceedings of the International Workshop on Security Protocols*, pages 69–87, 1997.

[119] D. Chaum. Blind signatures for untraceable payments. In *CRYPTO '82: Proceedings of Advances in Cryptology*, volume 82, pages 199–203, 1983.

[120] Masayuki Abe and Tatsuaki Okamoto. Provably secure partially blind signatures. In *CRYPTO '00: Proceedings of the 20th International Cryptology Conference on Advances in Cryptology*, pages 271–286, 2000.

[121] Ivan Osipkov, Eugene Y. Vasserman, Nicholas Hopper, and Yongdae Kim. Combating double-spending using cooperative P2P systems. In *ICDCS '07: Proceedings of the 27th International Conference on Distributed Computing Systems*, page 41, 2007.

[122] John R. Douceur. The sybil attack. In *IPTPS '01: Revised Papers from the First International Workshop on Peer-to-Peer Systems*, pages 251–260, 2002.

[123] Computing Bandwidth Adjustments. Tor Project. http://gitweb.torproject.org/tor.git?a=blob_plain;hb=HEAD;f=doc/spec/proposals/161-computing-bandwidth-adjustments.txt. November 2009.

[124] T. Narten and R. Draves. Privacy extensions for stateless address autoconfiguration in ipv6. http://tools.ietf.org/html//rfc3041, 2001.

[125] Relay Flags. Tor Project. http://git.torproject.org/checkout/metrics/master/out/dirarch/relayflags.csv. November 2009.

[126] The GMP Library. http://gmplib.org/.

[127] Claudia Diaz, Stefaan Seys, Joris Claessens, and Bart Preneel. Towards measuring anonymity. In *Privacy Enhancing Technologies*, 2002.

[128] Andrei Serjantov and George Danezis. Towards an information theoretic metric for anonymity. In Roger Dingledine and Paul Syverson, editors, *Proceedings of Privacy Enhancing Technologies Workshop (PET'02)*, pages 41–53, 2002.

[129] C. E. Shannon. A mathematical theory of communication. *SIGMOBILE Mobile Computing and Communications Review*, 5(1):3–55, 2001.

[130] Eugene Vasserman, Rob Jansen, James Tyra, Nicholas Hopper, and Yongdae Kim. Membership-concealing overlay networks. In *CCS'09: Proceedings of the 16th ACM Conference on Computer and Communications Security*, pages 390–399, 2009.

[131] Tor Path Specification. Tor Project. `http://gitweb.torproject.org/tor.git?a=blob_plain;hb=HEAD;f=doc/spec/path-spec.txt`. January 2010.

[132] The BitTorrent Protocol Specification. `http://www.bittorrent.org/beps/bep_0003.html`. January 2010.

[133] Tor Directory Protocol, Version 3. Tor Project. `http://gitweb.torproject.org/tor.git?a=blob_plain;hb=HEAD;f=doc/spec/dir-spec.txt`. January 2010.

[134] Rob Jansen, Paul Syverson, and Nicholas Hopper. Throttling tor bandwidth parasites. In *Proceedings of the 21st USENIX Security Symposium*. Internet Society, August 2012.

[135] Torservers.net. `https://www.torservers.net/`.

[136] The EFF Tor Challenge. `https://www.eff.org/torchallenge/`.

[137] Turning funding into more exit relays. `https://blog.torproject.org/blog/turning-funding-more-exit-relays`, July 2012.

[138] Paul Syverson, Gene Tsudik, Michael Reed, and Carl Landwehr. Towards an analysis of onion routing security. In Hannes Federrath, editor, *Designing Privacy Enhancing Technologies: International Workshop on Design Issues in Anonymity and Unobservability*, pages 96–114. Springer-Verlag, LNCS 2009, July 2000.

[139] David Chaum. Security without identification: transaction systems to make big brother obsolete. *Commun. ACM*, 28(10):1030–1044, October 1985.

[140] Robin Snader and Nikita Borisov. Eigenspeed: secure peer-to-peer bandwidth evaluation. In *Proceedings of the 8th international conference on Peer-to-peer systems*, IPTPS'09, pages 9–9, Berkeley, CA, USA, 2009. USENIX Association.

[141] Michael Luby and Charles Rackoff. How to construct pseudorandom permutations from pseudorandom functions. *SIAM J. Comput.*, 17(2):373–386, April 1988.

[142] E. De Cristofaro, C. Soriente, G. Tsudik, and A. Williams. Hummingbird: privacy at the time of twitter. In *33rd IEEE Symposium on Security and Privacy*. IEEE, 2012.

[143] Xu Cheng, Cameron Dale, and Jiangchuan Liu. Statistics and social network of youtube videos. In *In the Proceeding of the 16th IEEE International Workshop on Quality of Service (IWQoS)*, pages 229–238, 2008.

[144] Joan Feigenbaum, Aaron Johnson, and Paul Syverson. Probabilistic analysis of onion routing in a black-box model. In *Proceedings of the 2007 ACM Workshop on Privacy in Electronic Society (WPES 2007)*, pages 1–10, 2007.

[145] Michael Piatek, Tomas Isdal, Thomas Anderson, Arvind Krishnamurthy, and Arun Venkataramani. Do incentives build robustness in BitTorrent? In *Proceedings of the 4th USENIX Symposium on Networked Systems Design & Implementation (NSDI 2007)*. USENIX, April 2007.

[146] Thomas Locher, Patrick Moor, Stefan Schmid, and Roger Wattenhofer. Free Riding in BitTorrent is Cheap. In *5th Workshop on Hot Topics in Networks (HotNets), Irvine, California, USA*, November 2006.

[147] Rob Jansen, Kevin Bauer, Nicholas Hopper, and Roger Dingledine. Methodically Modeling the Tor Network. In *Proceedings of the 5th Workshop on Cyber Security Experimentation and Test*, August 2012.

[148] Cameron Kiddle. *Scalable network emulation.* PhD thesis, University of Calgary, 2004.

[149] S. Naicken, A. Basu, B. Livingston, S. Rodhetbhai, and I. Wakeman. Towards yet another peer-to-peer simulator. In *Proceedings of the 4th International Working Conference on Performance Modelling and Evaluation of Heterogeneous Networks (HET-NETs'06)*, 2006.

[150] S. Naicken, B. Livingston, A. Basu, S. Rodhetbhai, I. Wakeman, and D. Chalmers. The state of peer-to-peer simulators and simulations. *SIGCOMM Computer Communication Review*, 37(2):95–98, 2007.

[151] Shiding Lin, Aimin Pan, Zheng Zhang, Rui Guo, and Zhenyu Guo. Wids: an integrated toolkit for distributed system development. In *Proceedings of the 10th conference on Hot Topics in Operating Systems (HOTOS'05)*, 2005.

[152] The ns-2 Network Simulator. http://www.isi.edu/nsnam/ns/.

[153] The ns-3 Network Simulator. http://www.nsnam.org/.

[154] Scalable Simulation Framework, SSFNet. http://www.cc.gatech.edu/computing/compass/pdns/index.html.

[155] Sambuddho Chakravarty, Angelos Stavrou, and Angelos Keromytis. Traffic analysis against low-latency anonymity networks using available bandwidth estimation. In *Computer Security – ESORICS*, pages 249–267, 2010.

[156] John McLachlan and Nicholas Hopper. On the risks of serving whenever you surf: vulnerabilities in Tor's blocking resistance design. In *Proceedings of the 8th ACM Workshop on Privacy in the Electronic Society (WPES'09)*, pages 31–40, 2009.

[157] A. Demers, S. Keshav, and S. Shenker. Analysis and Simulation of a Fair Queueing Algorithm. In *ACM SIGCOMM Computer Communication Review*, volume 19, pages 1–12. ACM, 1989.

[158] Nagendra Modadugu and Eric Rescorla. The design and implementation of datagram TLS. In *Proceedings of the 11th Network and Distributed System Security Symposium (NDSS'11)*, 2004.

[159] The anonymizer. http://anonymizer.com/.

[160] P. Boucher, A. Shostack, and I. Goldberg. Freedom system 2.0 architecture. White paper, Zero-Knowledge Systems Inc., 2000.

[161] Elke Franz, Anja Jerichow, and Guntram Wicke. A payment scheme for mixes providing anonymity. In *TREC '98: Proceedings of the International IFIP/GI Conference on Trends in Distributed Systems for Electronic Commerce*, pages 94–108, 1998.

[162] Daniel R. Figueiredo, Jonathan K. Shapiro, and Don Towsley. Using payments to promote cooperation in anonymity protocols. Technical Report 03-31, University of Massachusetts, 2003.

[163] Philippe Golle, Kevin Leyton-Brown, Ilya Mironov, and Mark Lillibridge. Incentives for sharing in peer-to-peer networks. In *WELCOM'01: Proceedings of the Second International Workshop on Electronic Commerce*, pages 75–87, 2001.

[164] B. Cohen. Incentives build robustness in BitTorrent. In *Workshop on Economics of P2P Systems*, volume 6, 2003.

[165] Animesh Nandi, Tsuen-Wan "Johnny" Ngan, Atul Singh, Peter Druschel, and Dan S. Wallach. Scrivener: providing incentives in cooperative content distribution systems. In *Middleware'05: Proceedings of the ACM/IFIP/USENIX 2005 International Conference on Middleware*, pages 270–291, 2005.

[166] Ernesto Damiani, De Capitani di Vimercati, Stefano Paraboschi, Pierangela Samarati, and Fabio Violante. A reputation-based approach for choosing reliable resources in peer-to-peer networks. In *CCS '02: Proceedings of the 9th ACM Conference on Computer and Communications Security*, pages 207–216, 2002.

[167] Sepandar D. Kamvar, Mario T. Schlosser, and Hector Garcia-Molina. The Eigentrust algorithm for reputation management in P2P networks. In *WWW'03: Proceedings of the 12th International Conference on World Wide Web*, pages 640–651, 2003.

[168] Li Xiong and Ling Liu. PeerTrust: Supporting reputation-based trust for peer-to-peer electronic communities. *IEEE Transactions on Knowledge and Data Engineering*, 16:843–857, 2004.

[169] Runfang Zhou and Kai Hwang. PowerTrust: A robust and scalable reputation system for trusted peer-to-peer computing. *IEEE Transactions on Parallel and Distributed Systems*, 18(4):460–473, 2007.

[170] Adam Back, Ulf Möller, and Anton Stiglic. Traffic analysis attacks and trade-offs in anonymity providing systems. In *IHW '01: Proceedings of the 4th International Workshop on Information Hiding*, pages 245–257, 2001.

[171] Mathias Humbert, Mohammadhossein Manshaei, and Jean-Pierre Hubaux. One-to-n scrip systems for cooperative privacy-enhancing technologies. In *Proceedings of the 49th Annual Allerton Conference on Communication, Control, and Computing*, 2011.

[172] Michael Reed, Paul Syverson, and David Goldschlag. Anonymous connections and onion routing. *IEEE Journal on Selected Areas in Communications*, 16(4):482–494, 1998.

[173] David L. Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Communications of the ACM*, 24(2):84–90, 1981.

[174] George Danezis, Roger Dingledine, and Nick Mathewson. Mixminion: Design of a type III anonymous remailer protocol. In *IEEE Symposium on Security and Privacy*, 2003.

[175] George Danezis and Ian Goldberg. Sphinx: A compact and provably secure mix format. In *IEEE Symposium on Security and Privacy*, pages 269–282, 2009.

[176] Erik Shimshock, Matt Staats, and Nick Hopper. Breaking and provably fixing Minx. In *Proceedings of the 8th International Symposium on Privacy Enhancing Technologies (PETS'08)*, 2008.

[177] Matthew Wright, Micah Adler, Brian Neil Levine, and Clay Shields. The Predecessor Attack: An Analysis of a Threat to Anonymous Communications Systems. *ACM Transactions on Information and System Security (TISSEC)*, 4(7):489–522, November 2004.

[178] W. Brad Moore, Chris Wacek, and Micah Sherr. Exploring the potential benefits of expanded rate limiting in tor: Slow and steady wins the race with tortoise. In *Proceedings of 2011 Annual Computer Security Applications Conference*, December 2011.

[179] Jon McLachlan, Andrew Tran, Nicholas Hopper, and Yongdae Kim. Scalable onion routing with Torsk. In *CCS '09: Proceedings of the 16th ACM Conference on Computer and Communications Security*, pages 590–599, 2009.