Large-Scale Emulation of Anonymous Communication Networks

by

Sukhbir Singh

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2014

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Tor is the most popular low-latency anonymous communication system for the Internet, helping people to protect their privacy online and circumvent Internet censorship. Its low-latency anonymity and distributed design present a variety of open research questions related to — but not limited to — anonymity, performance, and scalability, that have generated considerable interest in the research community. Testing changes to the design of the protocol or studying attacks against it in the live network is undesirable as doing so can invade the privacy of users and even put them in harm's way. Traditional Tor research has been limited to emulating a few hundred nodes with the ModelNet network emulator, or, simulating thousands of nodes with the Shadow discrete-event simulator, both of which may not accurately represent the real-world Tor network. We present SNEAC (Scalable Network Emulator for Anonymous Communication; pronounced "sneak"), a large-scale network emulator that allows us to emulate a network with thousands of nodes. Our hope is that with such large-scale experimentation, we can more closely emulate the live Tor network with half a million users.

Acknowledgements

I would like to thank my supervisors, David Taylor and Ian Goldberg for their guidance, support, and patience. In spite of their busy schedules, David and Ian were always available to guide me and share their knowledge. I feel fortunate that I had the chance to work with them, and truly, I could not have wished for better supervisors.

I would also like to thank my thesis readers, Martin Karsten and Bernard Wong, for their time and feedback.

Finally, to the Cryptography, Security, and Privacy (CrySP) research group at the University of Waterloo, for all the wonderful discussions about security and privacy.

Dedication

To my father, who taught me everything. To Mom, KB, Shifa, for all the love.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Anonymity on the Internet and the Tor Network

The Internet serves as a platform for various forms of communication and activism allowing citizens to get their message across to a global audience almost instantly. The free flow of information and ideas made possible by the Internet has proved to be an effective medium for social change in many societies around the world [LGA+11].

Totalitarian regimes that were engaging in propaganda to further their own cause started realizing the threat the Internet posed to their rule — it was a decentralized and distributed system that connected people across the world who were sharing information over which governments had no control. To restrict the freedom of their citizens in the virtual world, many regimes started spying on the online activities of their citizens, while others started censoring the Internet by restricting access to or publication of content that they deemed inappropriate. Fortunately for the people, such attempts were unsuccessful and many systems for anonymous communication were developed that helped people maintain their privacy online and circumvent Internet censorship.

The most effective and widely used such system is called Tor (The Onion Router), which is based on a technique called *onion routing* [SGR97] designed at the U.S. Naval Research Laboratory in 1998. In onion routing, messages are repeatedly encrypted and sent across the network through several nodes called *onion routers*. Analogous to removing the layers of an onion, each onion router strips a layer of encryption and forwards the message to the next router; this process is repeated until the message reaches its final destination (see Figure 1.1). Tor is the most prominent implementation of onion routing [DMS04] and

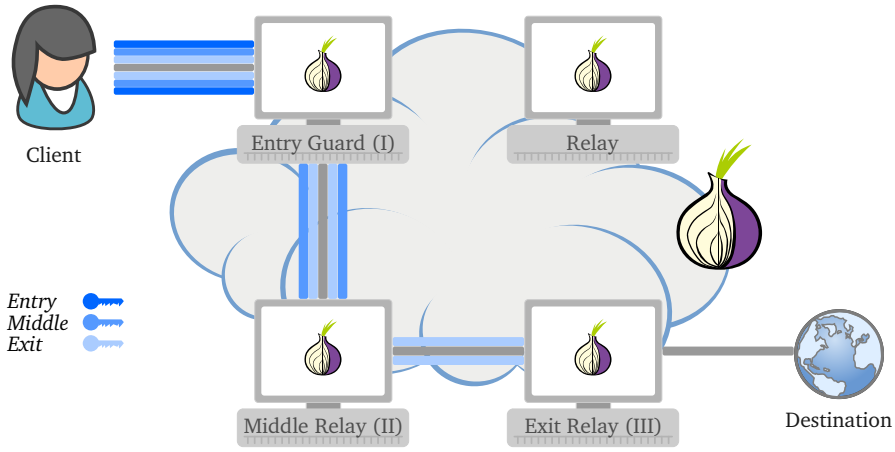

Figure 1.1: The Tor Network

has been in development since 2002 with about half a million users using it to protect their privacy on the Internet or to resist state-sponsored Internet censorship. Tor has thus far proved resilient to government attempts to de-anonymize its users or block its usage, even though the Tor developers and totalitarian regimes regularly engage in an arms race [Din11, Din12] to outdo each other. Figure 1.2 shows an increase in the number of users connecting to the Tor network from Turkey after the Turkish government censored Twitter in March 2014.

1.2 Tor Research

Tor's low-latency anonymity and distributed design present a variety of open research questions related to — but not limited to — anonymity, performance, and scalability, which have generated considerable interest in the research community, particularly among network and security researchers [Tor13]. As Tor is under active development, the research community frequently proposes design changes to the protocol, which may focus on improving the performance of the network or work towards implementing defences for the possible attacks against it. Testing these changes on the live network itself is undesirable as doing so can invade the privacy of users and in some cases, even put them in harm's way — any change made to the live network may impact all half a million users, some of whom trust Tor with their lives to keep them hidden under a cloak of anonymity as they access the Internet while living under the shadow of oppressive regimes.

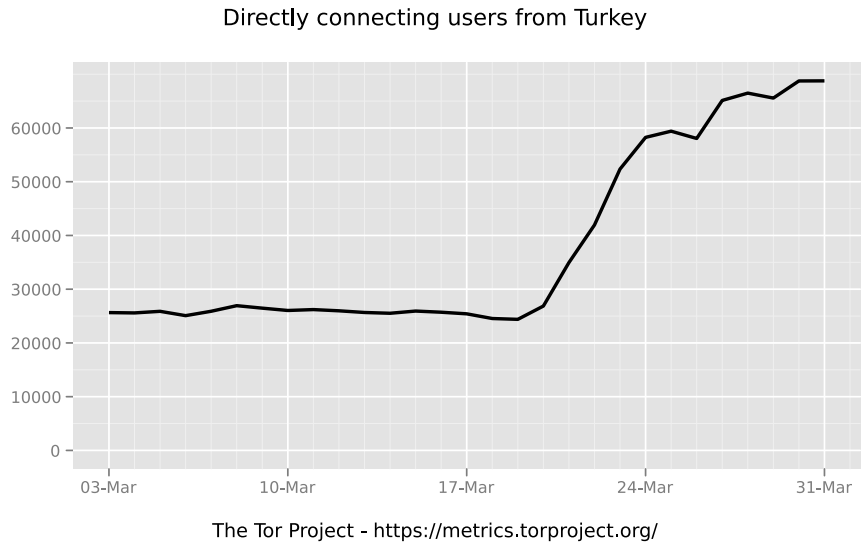

Figure 1.2: The number of users connecting to the Tor network from Turkey doubled after websites like Twitter were censored by the Turkish government in March 2014.

The two main critical points in the network for carrying out research are the volunteer-operated relays (*onion routers*) and the clients (*onion proxies*). Testing changes at either point presents different but related problems of putting users at risk or affecting the performance: relay operators are usually wary of running untested research code unless they feel assured that it brings a tangible benefit for the network, while for the clients, any change that may leak their identity puts them in harm's way.

As a workaround to this problem, Tor research has been performed outside the live network by setting up private offline Tor networks. Traditional ways of doing this involve setting up a local cluster with the ModelNet network emulator [BSMG11, VYW+02], running Tor on a single machine by simulating it with the Shadow discrete-event simulator [JH12], or by carrying out experiments on distributed testbeds like PlanetLab [CCR+03]. These approaches have their own strengths and weaknesses: while ModelNet provides for an emulated Tor network, it does not scale beyond a thousand nodes; Shadow scales to thousands of nodes [Sha14b] but it is a simulator and makes certain assumptions in order to simplify its simulation model; PlanetLab experiments also do not scale beyond a few hundred nodes and are difficult to reproduce. For the lack of a better tool, most literature on Tor has used one of these testing platforms to evaluate new design proposals or analyze attacks against the network, while being aware of their shortcomings and limitations. (We elaborate more

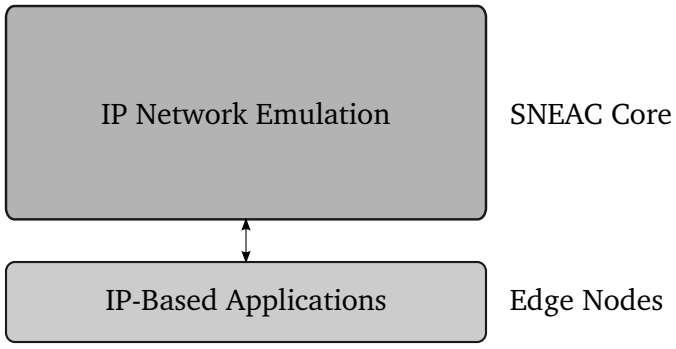

Figure 1.3: The two components of a SNEAC setup: the core and edge nodes.

on this in Chapter 2.)

An experimentation platform can usefully substitute for the real Tor network if it allows for the *emulation* of *thousands* of Tor nodes while ensuring that the results are independently verifiable and reproducible. ModelNet offers the realism of a live deployment of Tor, but falls short on scalability as it is not possible to run thousands of Tor nodes because of the limitations imposed by its FreeBSD-based emulator.

1.3 Our Contributions

We present SNEAC, the *Scalable Network Emulator for Anonymous Communication*, a network emulation testbed for Tor that runs real, unmodified Tor code and can scale to thousands of nodes. SNEAC is the only large-scale network emulation-based testbed that comes close to the real Tor network in both scale and realism of the experiments. This thesis makes the following contributions:

- The design and implementation of a scalable network emulator that derives from ModelNet's architecture but replaces its FreeBSD-based emulator with lightweight Linux Containers [LXC14] for the virtual routers and the Linux kernel's `netem` [Lin09] (network emulation) module for emulating link properties such as bandwidth, latency, packet loss, and jitter.

- A demonstration of the emulator's scalability by performing a large-scale experiment on the CrySP RIPPLE [Wat14] facility in which we run thousands of Tor nodes to study the mitigation of botnets using the Tor network — an experiment that could have only been performed by the emulation of a large number of nodes.

- Making a general-purpose, large-scale emulator available for the research community that can perform emulation of unmodified TCP- and UDP-based applications running on unmodified operating systems.

A SNEAC setup involves two components: the *core* and *edge nodes* (see Figure 1.3). The core machine performs the IP-level emulation while the edge node machines run unmodified IP-based applications. The functionality of both of these components is separated by design such that the core is oblivious to the applications running on the edge nodes — this modularization allows the SNEAC core to emulate *any* IP-based application, though in this thesis we will focus our attention on emulating Tor. To that end, we will discuss configuring and running an emulated Tor network on the edge nodes though the underlying process is the same for any other application.

SNEAC is open source, easy to configure (single-click setup), and runs on major Linux distributions without requiring modifications to the kernel. It can be downloaded from `https://crysp.uwaterloo.ca/software/sneac` and comes with documentation for setting it up on a cluster or running it on a single machine using virtual machines. We will be announcing SNEAC to the Tor research community and have plans to maintain and support its further development; our goal is to make SNEAC the *de facto* research platform for Tor experimentation.

The remainder of the thesis is organized as follows: in Chapter 2, we discuss related work in the field of Tor emulation and simulation. In Chapter 3, we describe the design of SNEAC followed by a discussion of its implementation in Chapter 4. We then demonstrate its scalability and efficacy in Chapter 5 by performing a large-scale experiment to study the effects of botnets using the Tor network. We conclude the thesis by discussing future work in Chapter 6 and summarizing our contributions in Chapter 7.

Chapter 2

Related Work

In this chapter we discuss the related work in the realm of Tor experimentation, which includes the *ModelNet* emulator, the *Shadow* simulator, the *PlanetLab* distributed testbed, and the *ns-3* network simulator.

2.1 ModelNet

ModelNet [VYW+02] is a network emulation testbed that allows distributed networks to be evaluated in Internet-like environments. A typical ModelNet setup consists of the emulator machine (called the *core*), which loads a network topology and emulates link characteristics like bandwidth, latency, packet loss, and jitter. One or more machines called *edge nodes* run the unmodified TCP- or UDP-based applications that have to be emulated and are connected to the core. The routing for the edge nodes is set up such that when a process on a given edge node wants to communicate with another process on the same or a different edge node, it goes through the core and experiences the link characteristics described above, thus emulating the transmission of a packet as it would travel across a real-world network. The emulator core is based on a custom FreeBSD 6.3 kernel running an improved version of `dummynet` [Riz97] to perform the emulation, while the edge nodes can run any unmodified operating system (typically Linux or FreeBSD).

The core is oblivious to the emulated applications running on the edge nodes since its job is to just emulate the network and route packets to their destinations. As such, the configuration of the processes on the edge nodes is independent of the core's configuration (except for the maximum number of processes that can be run, which we discuss later) but

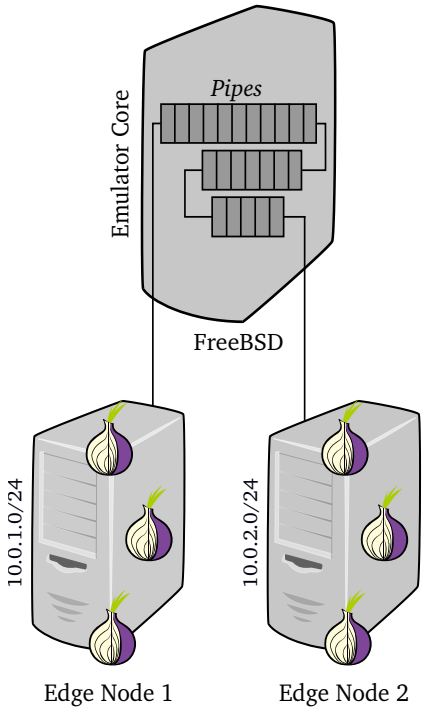

Figure 2.1: A ModelNet/ExperimenTor setup with one core and two edge nodes. Multiple Tor processes run on each edge node and the packets get routed through the core. The *pipes* in the core denote emulated links with each of them having an associated packet queue and queuing discipline.

is tailored to the application that has to be emulated. Bauer *et al.* [BSMG11] designed *ExperimenTor*, a toolkit for emulation of the Tor network on the ModelNet testbed that sets up the Tor processes on the edge nodes by configuring the directory authorities, relays, and clients, and comes with tools that automate testing the emulated network.

ModelNet supports multiple cores and multiple edge nodes, though most ExperimenTor deployments involve a single core and a single edge node which can run at most a thousand Tor nodes. The scalability of the testbed is restricted both by the resource constraints of the FreeBSD 6.3 kernel and by the complexity involved in setting up a ModelNet cluster.

- ModelNet runs (only) on a custom FreeBSD 6.3 kernel, which has limitations on the amount of resources it can access, such as the number of CPU cores it supports, the physical memory it can address, and the supported capacity of the NICs (network interface cards).

- It is also no longer maintained — the last release was in 2005 [Mod05] and the modified FreeBSD 6.3 kernel it runs on reached its end-of-life in 2010 [Fre14].
- The software has some critical bugs, such as loading an incorrect topology induces kernel panics, which makes it problematic to use for running even the simplest of experiments.
- The original documentation is lacking and even though attempts have been made to document the installation and configuration process [GB13], setting up and running an experiment on ModelNet is difficult to get right because of the various complexities involved, such as compiling a FreeBSD kernel and making configuration changes, all in the absence of any diagnostic messages.

For the reasons discussed above, most experiments performed with ModelNet have been restricted to a single edge node and a single core, running approximately a thousand Tor nodes. Other researchers have used simulators such as Shadow (Section 2.2) to get around the scalability limits and intricacies involved in running an experiment on ModelNet, at the cost of sacrificing the realism ModelNet offers by virtue of its emulation of unmodified Tor processes running in real time.

2.2 Shadow

Shadow [JH12] is a discrete-event simulator that runs a private Tor network on a single machine and scales to thousands of nodes. It simulates the network layer but links to and runs real application code by encapsulating it in plug-in wrappers (library *shims*) that provide the necessary functions for Shadow to interact with the application.

Shadow runs multiple virtual nodes by keeping a copy of all variable application state for each node in the simulation and, when it has to execute an event in the application, it does a *context switch* that copies the state information from its memory to the physical memory location where the application expects the state to exist. The control then passes to the application and when it returns, the updated state is copied back into Shadow's memory and the context is switched back to it. For library calls such as `socket`, Shadow intercepts and re-routes them to their simulated counterparts using function interposition (`LD_PRELOAD`), allowing applications to run in a simulated environment without requiring changes to their code.

Shadow can simulate large and diverse Tor networks (thousands of nodes) on a single machine [Sha14b], thus overcoming the limitations of the ModelNet emulator. These scalability benefits that Shadow offers come at a cost — by virtue of its being a *simulator*, the validity of the results depends on the abstract model it is based on, which may not accurately represent or encompass all low-level system interactions.

- Simulation is less accurate than emulation as simulators make assumptions about various system processes in order to simplify their simulation model and they also reinvent the wheel for the most basic primitives.
- Shadow implements its own version of the `socket` library and network protocols like TCP and UDP. While this allows Shadow to simulate unmodified applications (by re-routing standard library calls to their virtual counterparts), it creates an extra layer of complexity, as rewriting networking stacks from scratch is difficult to get right; indeed, Shadow has seen bugs in its TCP implementation [Sha14a] that can drastically affect its experimental results.
- To save CPU cycles and simulation time, Shadow preloads (function interposition) certain cryptographic functions (`AES_encrypt`, `AES_decrypt`, `EVP_Cipher`). When preloading these functions, Shadow does not perform encryption and decryption as it assumes that applications running during simulation do not require confidentiality. While doing this saves CPU cycles and simulation time, performing cryptographic processing is an important part of Tor's functionality and skipping any such primitive should be avoided.
- Shadow's scalability is restricted to the resources available on a single machine as it is not possible to run a single instance of Shadow across multiple machines. While this is ideal for running experiments that scale within the available resource limits, Shadow cannot make efficient use of computing resources (such as an experimentation cluster) beyond what it can access from the machine it runs on.

As compared to ModelNet, Shadow offers benefits such as scalability and ease of setup, but experiments run on Shadow offer less accuracy than the ones run on ModelNet. In spite of that, much of the recent research on Tor has used Shadow for experimentation because of the benefits it offers and for the lack of a better emulation platform.

2.3 PlanetLab

PlanetLab [CCR+03] is a global, distributed testbed for network and distributed systems research with nodes spread all across the globe. Even though a substantial amount of Tor research has been carried out on PlanetLab, recent work has moved towards using Shadow or ModelNet for a variety of reasons.

- The most common issue with running experiments on PlanetLab is that the results are generally not reproducible [SPBP06] because they depend on the condition of the PlanetLab network, which may vary over time. To generate statistically valid results, experiments have to be run for long periods of time while ensuring availability of sufficient resources, which may not be always possible.
- The scalability of experiments is restricted as PlanetLab has around twelve hundred available nodes (at the time of writing) out of which not all are usable because they are shared with researchers from all over the world.
- PlanetLab deployments are difficult to control and it is not possible to adapt them to the network characteristics of the live Tor network.

Other related distributed emulation testbeds, such as DETER [DET14] and Emulab [Emu14], are subject to the same limitations — shared resources which may skew experimental results and limited scalability because of a small number of available nodes.

Chapter 3

Design

3.1 Design Goals

The decisions behind the design of SNEAC were influenced by the goal of developing an experimentation platform for network research that would allow for the *emulation* of a large number of application nodes limited only by the physical resources of the experimentation platform. The design satisfies the following requirements:

- **Emulation**: Emulation-based testbed that runs real, unmodified application code.
- **Scalability**: Scales to thousands of application nodes.
- **Accuracy**: Experimental results are independently reproducible and verifiable.
- **Usability**: Open source, easy to manage and control with single-click setup.

3.2 Architecture

The design and architecture of SNEAC is based on that of ModelNet — one or more machines called *edge nodes* run the unmodified TCP- or UDP-based applications that have to be emulated and are connected to a single machine called the *core*, which loads a network topology and emulates link characteristics like bandwidth, latency, packet loss, and jitter. The routing for the edge nodes is set such that when a process on a given

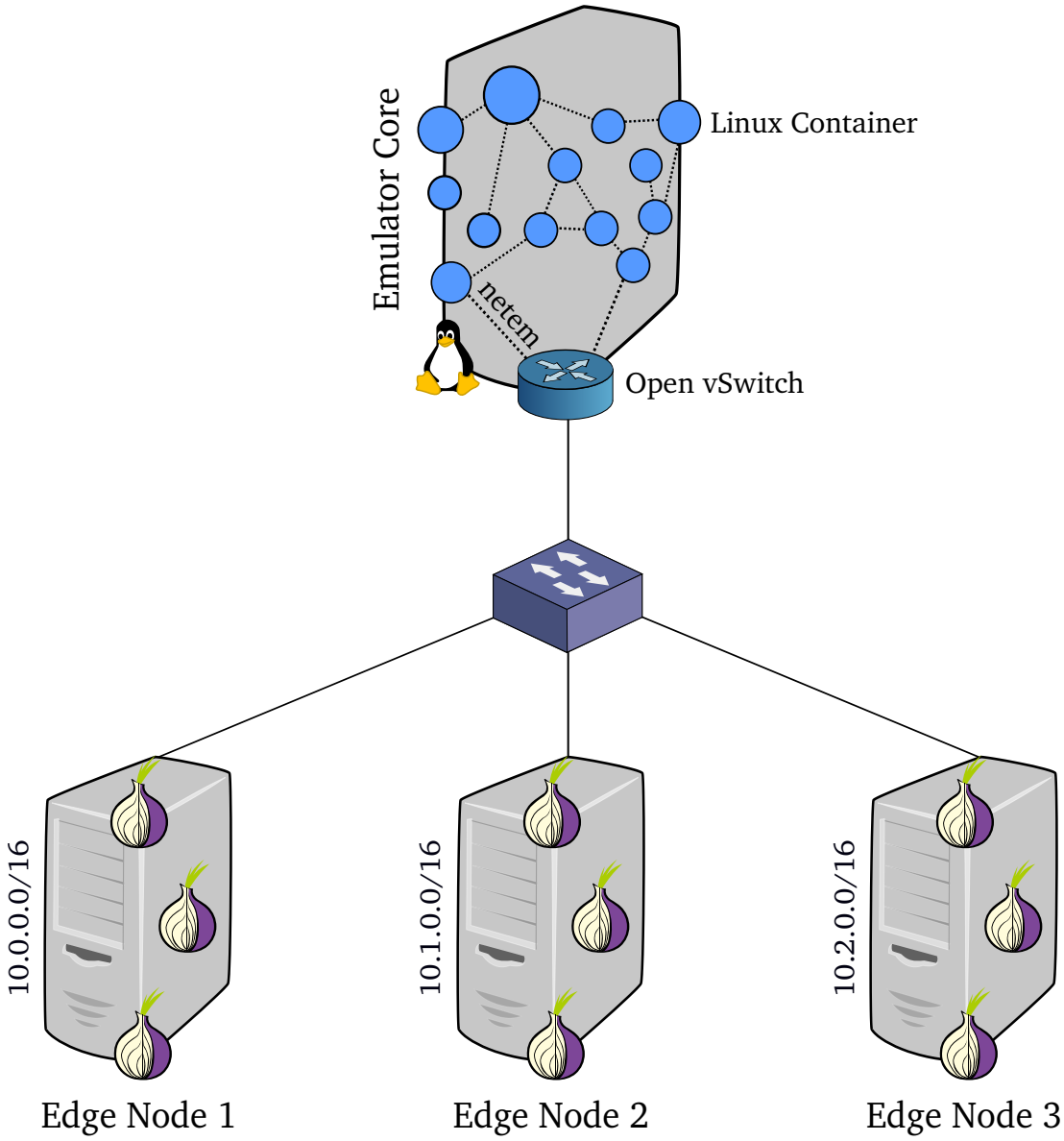

Figure 3.1: Architecture of a SNEAC setup being used to emulate Tor.

edge node wants to communicate with another process on the same or a different edge node, it goes through the core and experiences the link characteristics described above, thus emulating a real-world network. Figure 3.1 shows the architecture of a SNEAC setup with one core machine and three edge nodes.

The SNEAC core runs an unmodified Linux installation with lightweight LXCs (Linux Containers) acting as virtual routers, `netem` performing the link emulation, and Open vSwitch [PPA+09] routing packets to and from the edge nodes. For setting up the containers and configuring the link characteristics, we use a custom version of Mininet [LHM10], described below.

An alternative to the SNEAC core is the `ns-3` [ns-14] discrete-event network simulator. Even though it is primarily intended for network simulation, it is possible to use `ns-3` as a real-time network emulator by connecting its simulation core to a device operating in real time; thus, applications running on a real device connected to the core will run in real time but the network layer will still be *simulated*. However, we do not use `ns-3` for the core and base our implementation on Mininet using LXCs.

The edge nodes run any unmodified operating system such as Linux or FreeBSD and the applications that have to be emulated also run unmodified.

It is important to note that even though the focus of this thesis is on emulating Tor relays and clients, SNEAC can emulate *any* TCP- or UDP-based application, unmodified. This is because the emulation core is oblivious to the configuration of the applications on the edge nodes — its job is simply to emulate the network and re-route the packets.

We will now discuss the various components of the emulator and the edge nodes.

3.3 The Emulator Core

The emulator performs two main tasks: *emulation*, handled by the Linux kernel's `netem` module, and *routing*, which involves LXCs acting as virtual routers and Open vSwitch routing packets between the edge nodes and the LXCs. Before it can do any of that, however, it needs to load and set up a virtual topology with routers and links and their associated characteristics such as bandwidth, latency, packet loss, and jitter. We do this using our custom version of the Mininet emulator with many scalability enhancements, which we discuss in detail below.

Topology	Hosts	Switches	Links	Startup (sec)	Shutdown (sec)
Linear (100)	100	100	199	48 ± 4	20 ± 2
Linear (200)	200	200	399	100 ± 30	30 ± 10
Linear (300)	300	300	599	180 ± 7	49 ± 2
Linear (400)	400	400	799	330 ± 6	70 ± 2
Linear (500)	500	500	999	890 ± 50	220 ± 20

Table 3.1: Startup and shutdown time for various topologies on Mininet 2.1.0+ with Open vSwitch kernel switches (version 2.1.0) and the POX 0.2.0 controller running on a Linux 3.11.0 installation.

3.3.1 Mininet

Mininet [LHM10] is a network emulator for rapid prototyping of software-defined networks (SDN) that sets up a collection of end-hosts, switches (or routers), and links on a single machine, allowing users to quickly deploy custom networks with hundreds of nodes. A Mininet-based network is made up of the following components:

- **Hosts**: Hosts are lightweight LXCs and are essentially a group of user-level processes (*bash*) moved into a network namespace that provides them with their own network devices, routing tables, and firewall rules.
- **Links**: A link is a virtual Ethernet pair (*veth pair*) which connects two virtual interfaces and is bidirectional in nature. The data rate of the link is enforced by Linux Traffic Control (`tc`) and other characteristics such as latency, packet loss, and jitter are handled by the `netem` module.
- **Switch**: Mininet uses either the default Linux bridge or Open vSwitch (running in kernel space) to switch packets across interfaces.
- **Controller**: A controller manages the flow of packets between switches in a software-defined network. Mininet supports local as well as remote controllers.

While Mininet is ideal for emulating small networks with a few hundred nodes, it has serious scalability issues when emulating networks with thousands of nodes, partly because of its reliance on components such as Open vSwitch and partly because of design choices that were not made with scalability in mind. One such limitation is that the Mininet process is *single-threaded*: all operations, such as adding the LXCs or creating the *veth pairs* are performed sequentially. While this hardly makes a difference for running small

Node	Startup (sec)		Shutdown (sec)	
	LXC	Switch	LXC	Switch
Linear (100)	7.8 ± 0.8	19 ± 1	1.9 ± 0.2	6.1 ± 0.4
Linear (200)	17 ± 1	90 ± 4	4.7 ± 0.5	18 ± 1
Linear (300)	26 ± 1	220 ± 10	7.9 ± 0.5	28 ± 1
Linear (400)	36.5 ± 0.4	420 ± 40	11.1 ± 0.2	39 ± 1
Linear (500)	54 ± 4	800 ± 100	17 ± 1	40 ± 5

Table 3.2: Startup and shutdown time comparison between host-based and switch-based routing for vanilla Mininet. The topologies include nodes of only a particular type (either LXC or switch) with an equal number of links for a given topology.

topologies, as the network size increases, it becomes apparent that this is an inefficient approach especially since the parallel creation of hosts and interfaces is supported by the kernel.

By far the biggest bottleneck, however, comes from Open vSwitch, which Mininet uses for the virtual routers: as the number of Open vSwitch switches increases, the performance decreases and the startup/shutdown time increases, with this aggravated further as more and more ports are added to the switches; these limitations are documented in the Open vSwitch manual [Ope14] which puts an upper limit on the supported number of switches per OVS instance at 256.

These scalability issues arise because Open vSwitch was not designed to support a large number of switches on a single machine; indeed, it is a virtual switch that is used in hardware virtualization environments but finds its use in Mininet (even though it was not designed for that specific purpose) because it supports OpenFlow [MAB+08] and because its kernel-space switching provides better performance than the other available options. Table 3.1 benchmarks the startup and shutdown times (repeated 50 times) for various topologies on a Mininet 2.1.0+ installation with Open vSwitch kernel switches (version 2.1.0) running on Linux 3.11.0. As we can observe, running a topology with thousands of nodes is likely not feasible as the startup time increases superlinearly in the number of switches or links, though starting hosts is sufficiently fast as it is being done inside the kernel. Even assuming that a topology finishes setting up, it is unlikely that Mininet and hundreds of Open vSwitch switches will be able to handle the traffic without affecting the performance and validity of the experiment.

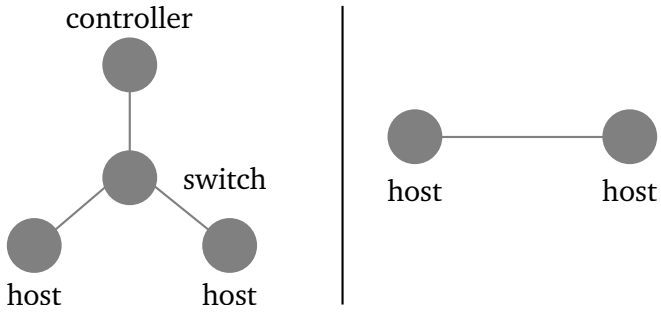

Figure 3.2: Architecture of a Mininet setup (left) compared with that of SNEAC (right).

3.3.2 Mininet Turbo

To scale Mininet to thousands of nodes, we make performance improvements to its code and architecture and call our fork *Mininet Turbo*. We make the following changes to Mininet:

- **LXC-Based Routing**: As the link characteristics are set on the *veth pairs* and *not* the switches or hosts, it is possible to replace the Open vSwitch *switches* with *hosts* for the virtual routers. A host (LXC) in this case acts as a simple packet forwarder, routing packets based on its routing table. As shown in Table 3.2, hosts scale better than switches, are faster to start up and shut down (repeated 50 times), and by doing this we remove the Open vSwitch bottleneck as LXC creation, configuration, and routing (described below) are handled by the kernel. Also, there is no artificial scalability limit as the number of LXCs is effectively limited only by the physical resources of the machine.

- **Static Routing**: Another advantage of using LXCs for routing is that we *do not* have to use any controller — static routing is set up between LXCs using the `ip route` command as each LXC has its own routing table. Mininet uses a controller to do software-defined networking while the SNEAC core is only concerned with IP-level network emulation and therefore does not need a controller. Using static routing results in a significant improvement in performance as the routing is now handled by the kernel, as compared to using a controller or setting Open vSwitch flow rules for each switch.

- **Threading**: Instead of creating hosts and links sequentially, we use threads, which significantly speeds up the time it takes to set up a topology.

At this stage, a significant difference between Mininet and our implementation can be explained: unlike Mininet, in our emulator, *no* traffic is generated on the emulator machine. This is different from Mininet where hosts that generate the packets and switches that route them all share the resources of a single machine. In SNEAC, the hosts that generate the traffic run on the edge nodes and consume the resources (such as CPU, physical memory) of their respective edge node machines, while the emulator only performs the emulation and routing, which not only helps us gain in performance but also allows us to leverage the resources of multiple machines.

3.3.3 LXC-Based Routing

In SNEAC, there are two types of nodes: *client* and *non-client*. A client is a node analogous to a user connecting to a gateway router (ISP) while a non-client node is modeled after an autonomous system (AS). This gives rise to an important distinction in the way they are set up — a client node can receive packets from the outside world (edge nodes) as they are connected to a node in the *root* network namespace, a namespace to which physical devices can be assigned.

We need to handle multiple links for a client node because *veth pairs* are intrinsically bidirectional and it is not possible to have asymmetric link characteristics like bandwidth, which is not an accurate representation of a node connecting to an ISP in the real world. Such a node may have different up/down bandwidths and packets sent between the node and the ISP will have lower latency than the packets sent to other networks. We model this behaviour by using multiple links between the client and the *root* node.

Figure 3.3 shows a client node connected to a *root* node with three different types of links: *down*, *up*, and *self*. Packets that are incoming to a host (with a specific down bandwidth) traverse through the *down* link while outgoing packets (with an up bandwidth) traverse the *up* link. The *self* link is for packets that have the same source and destination as that of the host, such as for packets sent between a client and its gateway router (ISP).

Taking the above example again, assume that the host will handle all packets in the `10.0.0.0/16` address space. Packets from the `10.1.0.0/16` network destined to this host will come through the *down* link and go back through the *up* link, with varying bandwidths and other characteristics. However, packets sent with the same destination as that of the host (`10.0.0.0/16`) will traverse the *self* link and only experience latency and packet loss.

To route packets between the *root* node and the other nodes, we use Open vSwitch which acts as a bridge connecting all client nodes to the *root* node. (We point out that this is the proper use case of Open vSwitch since this is what it was designed to do — function

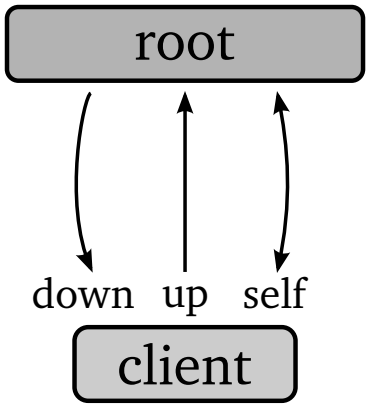

Figure 3.3: A client node connected to a node in the *root* network namespace has three different types of links to handle asymmetric bandwidth and latency.

as a virtual switch.) An Open vSwitch bridge is created to which all the interfaces (all client node links) and a given physical interface are attached. Flow rules are then installed based on the source and destination fields so that packets can be routed to the appropriate nodes where their own routing table routes them further to other client nodes. Open vSwitch handles the flow of packets only at the edges of the emulator and ensures that incoming and outgoing packets respect the *down* and *up* links.

3.3.4 Link Attributes: `tc` and `netem`

The actual emulation of the network is performed by Linux Traffic Control (`tc`) and the `netem` module, which set link characteristics for the *veth pairs* like bandwidth, latency, packet loss, and jitter. For a link, the bandwidth is set in Mbits/sec, latency and jitter in `ms`, and the packet loss in percentage.

With LXC-based routing and using kernel modules like `tc` and `netem`, we move away from external dependencies and closer to the kernel, which not only helps us scale better but assures us of the validity of our emulator as the emulation is performed by standard Linux network emulation kernel modules.

3.4 Edge Nodes

The edge nodes run unmodified TCP- or UDP-based applications that have to be emulated on unmodified operating systems, which are typically Linux or FreeBSD. Though the edge nodes require no special configuration, applications have to be configured so that we can run multiple applications on a single machine while ensuring that all of their routing takes place through the emulator. The general configuration process applies to any application, with some difference in application-specific details. Here, we focus our attention on emulating Tor.

We start by creating an *interface alias* for each application we want to run (such as `eth0:1`, `eth0:2`) and then binding applications to these *interface aliases*. This is done by intercepting the `bind()` and `connect()` calls to use the IP address of an alias as the local address; before launching an application, we call `LD_PRELOAD` on it and set its source IP to be one of the IP addresses of the aliases. This way, we can bind multiple processes to multiple IP addresses all running on the same machine, such that packets destined for a particular IP address reach the right process.

Another related issue is that for two processes running in the same subnet, the kernel routes the packets between them through the loopback interface instead of sending them out through the default gateway. As an example, if two applications are running on `10.0.0.1` and `10.0.0.2` respectively, packets from `10.0.0.1` to `10.0.0.2` will go through the loopback interface and not through the default route for `10.0.0.0/8`. To fix this, we flip the 23rd bit of the destination IP address field using `LD_PRELOAD` (a hack borrowed from ModelNet), so that the destination address of `10.0.0.2` becomes `10.128.0.2`; the kernel sees that this packet is destined for another subnet and it sends it out through the default route for `10.0.0.0/8`, which is set to be the emulator. The emulator then flips the bit back and sets the bits in the source address so that the routing process is seamless and the applications remain unaware of this.

To set up the Tor processes on the edge nodes, we use Chutney [Tor14a], which sets up and configures a private, test network by configuring Tor directory authorities, relays, and clients. Our changes to Chutney include adding support for configuring the nodes so that they route their packets through the emulator (using the `LD_PRELOAD` trick described above) instead of doing it through the loopback interface, and scaling it to work with more than 255 Tor nodes. To emulate a live Tor network, we have one set of directory authorities (usually three) running for the entire experiment on a given edge node, while the relays and clients are spread across all edge nodes. We use a Tor controller [Tor14b], which is a program used for interacting with a locally running Tor process, to control the nodes

once they have been set up, allowing us to manage our experiments locally and easily. It is possible to emulate any version of Tor, unmodified, just by installing a local copy of it.

As we have mentioned before, the SNEAC core is a general-purpose network emulator that can emulate *any* TCP- or UDP-based application; the only configuration requirement is that the applications are started with the `LD_PRELOAD` library discussed above, which binds the applications to a specific IP address and flips the required bits.

Chapter 4

Implementation

The core implementation of SNEAC is in Python because we use the Python API provided by Mininet to create and configure a virtual network. We base SNEAC on a custom version of Mininet which we call *Mininet Turbo* (as described in Section 3.3.2), which adds support for multi-threading, LXC-based routing, and configuring Mininet to talk to the outside world. There are various other components involved in setting up an experiment which we describe below.

4.1 Emulator Core

We start by loading a network topology from a GraphML file that has information about nodes (vertices) and links (edges) with related attributes such as bandwidth, latency, packet loss, and jitter. We follow the format used by Shadow for our experiments but it is possible to use other topologies such as those provided by the Internet Topology Zoo [KNF+11].

Using this topology information, we create a virtual network by calling Mininet Turbo, which configures the LXCs, creates the links (*veth pairs*) and applies the link attributes. This is a multi-step process:

- Once the topology has been loaded into memory, Mininet creates the LXCs and sets up the *veth pairs* between them, applying `tc` and `netem`. At this stage, the topology is isolated with links only between the LXCs and no communication with the outside world.

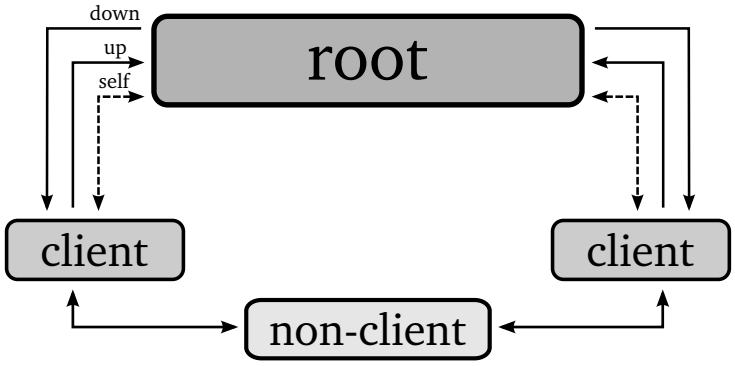

Figure 4.1: Layout of client and non-client nodes with their respective links in a SNEAC setup. The links between *client* and *non-client* nodes are symmetric.

- We then create a node in the *root* network namespace to which *client* nodes can be attached and physical devices can be assigned. For each node in the topology that is a *client* node, we create three links between it and the *root* node to handle asymmetric bandwidths as described in Section 3.3.3.

- At this stage, all *client* and *non-client* nodes have been configured and the links between them and the *root* node have been created. Now we can set up the routing among all nodes, whether *client* or *non-client*, taking into consideration that traffic can only enter or exit from the *client* nodes.

- Static routing is set up between the LXCs based on *shortest path* using the latency as the weight (configurable). We start by associating each node in the network to a particular subnet; the node will only handle packets that have the destination address within the subnet the node is configured to handle the packets for. We then pairwise iterate over all nodes and calculate the shortest path between them, setting up the routing table for each node in the path.

  The routing is configured using the `ip route` command for each LXC as LXCs have their own routing table. An example routing entry looks like

  ```
  ip route add 10.0.0.0/16 via 192.168.0.1 dev h1-eth0
  ```

  This tells an LXC that packets it receives destined for the `10.0.0.0/16` address space should be sent to the gateway at `192.168.0.1` through the `h1-eth0` interface.

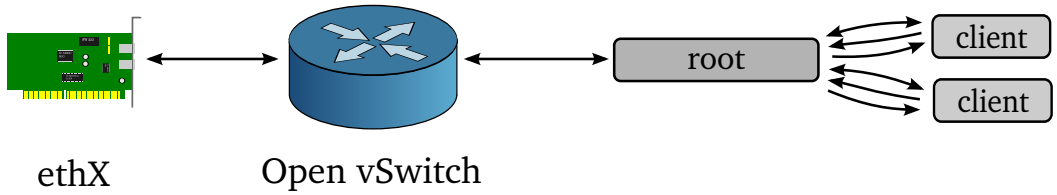

Figure 4.2: Open vSwitch creates a bridge between the physical interface (`ethX`) and the *root* node, allowing packets to be delivered to the *client* nodes.

- To handle packets incoming to the *client* nodes, we have to configure the routing at the *root* node. This routing should respect the three different links a *client* node has by making sure that incoming traffic goes through the *down* link, outgoing through the *up* link, and packets having the same source and destination traverse the *self* link (see Figure 4.1).

  To do this, we create an Open vSwitch bridge to which we attach all the *client* node links and a given physical interface, such as `eth0`, as shown in Figure 4.2. We then install flow rules that send the packets to the appropriate interface based on their source and destination addresses and also perform other tasks such as rewriting the MAC addresses to match that of the edge node from which the packets are coming and to the client node they will first talk to. An example flow rule looks like

  ```
  ovs-ofctl add-flow br0 \
      "in_port=1, ip, \
      nw_src=10.0.0.0/22, nw_dst=10.128.0.0/22,
      actions=load:1->NXM_OF_IP_SRC[23], load:0->NXM_OF_IP_DST[23], \
              mod_dl_dst:3a:21:d6:df:b1:87, \
      output:4"
  ```

  This flow rule routes IP packets coming into port `1` (the physical interface, `ethX`) of the bridge `br0` with the source address of `10.0.0.0/22` and destination address of `10.128.0.0/22` to port `4` (down link for the first LXC). The 23rd bits of the source and destination fields (refer to Section 3.4 for the bit flipping) are flipped, and the destination MAC address of the packets is changed to match that of the down interface (port `4`) of the first LXC, to which the packets are delivered. The LXC then accesses its routing table and forwards the packets to the appropriate router.

After the above steps have finished, the core is ready to emulate the network and route

the packets; *Mininet Turbo* has set up the topology and is no longer relevant at this stage because all emulation and routing is handled by the kernel and Open vSwitch.

4.2 Edge Nodes

The configuration of the edge nodes involves starting the application processes (Tor) and configuring the routing for them. The core needs a list of the edge nodes (their IP addresses) that will be running so that it can configure the routing. There is no limit on the number of edge node processes that can be run provided the emulator can handle the traffic, and that the number stays within the address space the emulator is configured to handle packets from — if an emulator is configured to support the `10.0.0.0/16` address space, the maximum number of supported edge node processes will be 65534. It is important to note that this is different from ModelNet where the maximum number of supported edge node processes *is equal* to the number of routers in the topology; there is no such restriction in SNEAC and it is possible to run a network with a few hundred routers but thousands of edge node processes.

We use Chutney to configure the Tor nodes; it creates the directory authorities', relays', and clients' configuration files and starts the Tor processes. This step also creates an equal number of interface aliases to which we bind the Tor processes using `LD_PRELOAD` library *libipaddr* (shipped with ModelNet), which binds the Tor processes to the interface aliases and flips the bits when sending out packets (refer to Section 3.4).

Once the Tor nodes have started, no further configuration is required and the nodes can be controlled using a Tor controller. To monitor the performance of the network, one can start an HTTP server and use the Tor client to download files from it or, for fine-grained information about the Tor processes, one can monitor them as one normally would by using a controller.

Chapter 5

Experiments

We will now demonstrate SNEAC's scalability by emulating thousands of Tor nodes running on multiple edge nodes. We also show its fidelity by comparing it against ModelNet, showing that for smaller experiments (limited by ModelNet), SNEAC's performance is statistically equivalent to that of ModelNet.

5.1 Experimental Setup

We deployed SNEAC on the CrySP RIPPLE facility [Wat14], which is an experimentation platform for research into large-scale privacy enhancing technologies.

- **Hardware**: For our experiments, we used seven edge node machines (called *ticks*; `tick[0-6]`) connected to a single emulator core machine (called *tock*). Each `tick` machine has 80 CPU cores (160 Hyper-Threaded), 1 TB of physical memory, and four 40 Gigabit Ethernet NICs. The `tock` machine has 80 CPU cores (160 Hyper-Threaded), 2 TB of physical memory, and four 40 Gigabit Ethernet NICs. On all the machines, we use interface bonding to bond the 40 GbE NICs into a single interface for a combined speed of 160 Gbps for each bonding interface per machine, limited to 128 Gbps by the system bus.
- **Software**: All machines in the cluster run Ubuntu 12.04.4 LTS with Linux kernel 3.13.0. The core machine runs Open vSwitch version 2.1.0 and our improved version of Mininet (*Mininet Turbo*) based on Mininet 2.1.0+. On the edge nodes, we run `tor-0.2.5.1-alpha` which uses the newer `ntor` handshake [Pro11], and

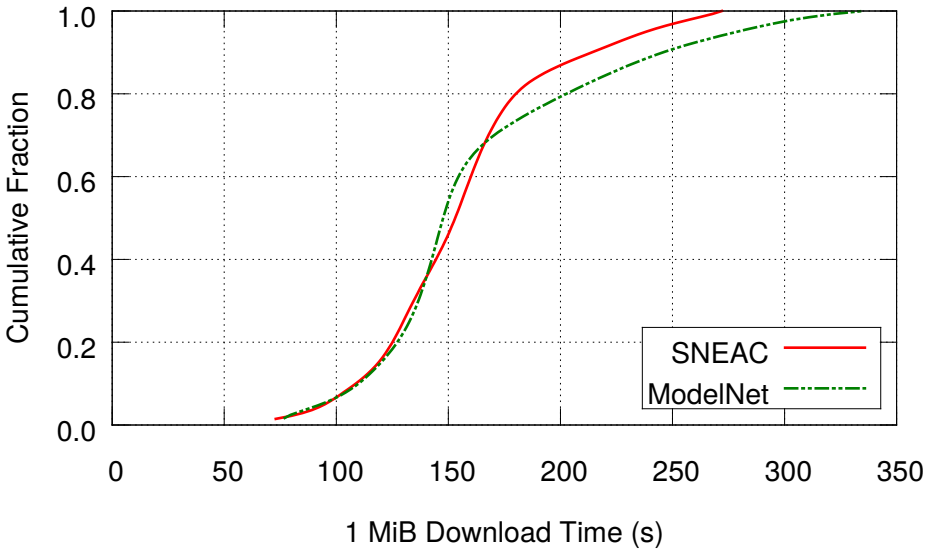

Figure 5.1: Download times for a 1 MiB file over Tor on SNEAC and ModelNet.

`tor-0.2.3.25` which uses the older TAP handshake [DMS04]. (We will talk more about the difference between these handshakes in Section 5.2.)

5.1.1 Comparison with ModelNet and Mininet

We start by comparing the performance of SNEAC with ModelNet and Mininet by running a Tor performance test and an `iperf` TCP throughput test.

ModelNet

Our experimental testbed consists of a single core machine (Linux for SNEAC; FreeBSD for ModelNet) connected to one edge node (Linux for both SNEAC and ModelNet). We use a network topology consisting of 36 nodes (routers) and 40 edges (links); each link in the topology has a bandwidth of 500 Kbps, latency of 10 ms, and queue length of 10 packets. We then configure a Tor network with three directory authorities, four relays, and seven clients using Chutney. To measure the performance of the network, we start the seven clients concurrently and download a 1 MiB file from a local HTTP server (a standard Tor performance test [Kar09]); we note the time it takes for all clients to download the file with

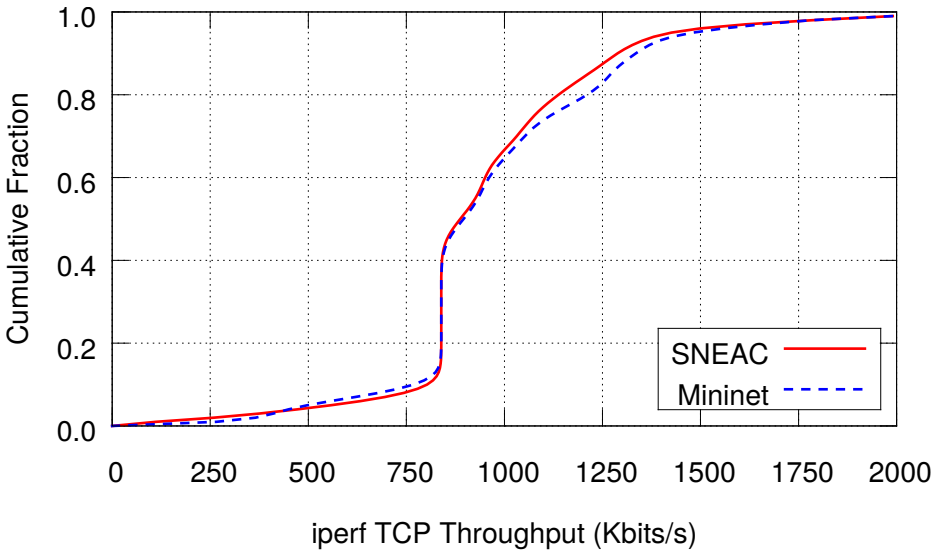

Figure 5.2: Network performance tests with `iperf` on SNEAC (with LXC-based routing) and Mininet (with Open vSwitch switch-based routing).

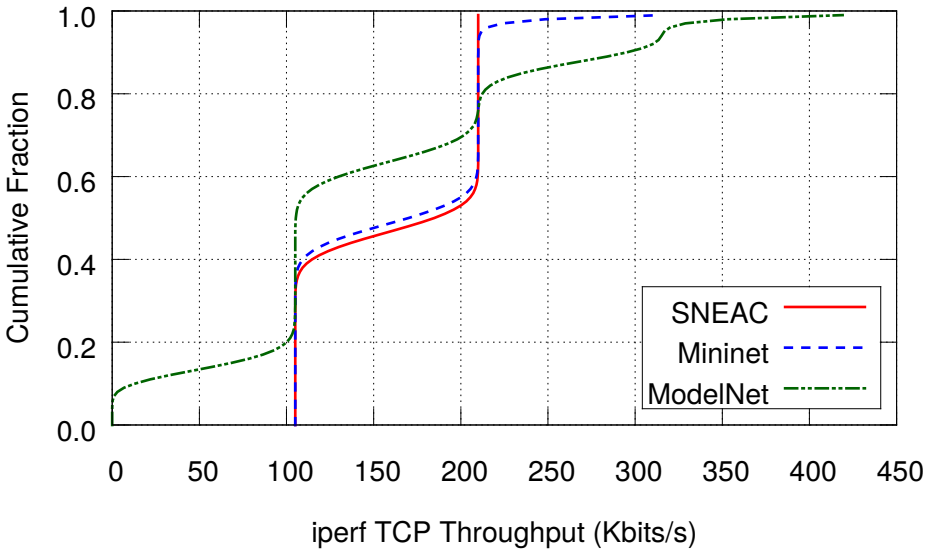

Figure 5.3: Performance comparison between SNEAC, Mininet, and ModelNet.

the experiment repeated 25 times. Figure 5.1 shows the download times for all seven clients on the given topology, from which we note that experiments run with SNEAC produce the same results as those run with ModelNet for the majority of the test runs. We found in our experiments that the performance of ModelNet varies between subsequent runs, making it difficult to accurately compare it with SNEAC.

Mininet

Mininet uses Open vSwitch switches for the routing, which we replace with lightweight LXCs in SNEAC. This is possible because link attributes such as bandwidth, latency, packet loss, and jitter are set on the *links* (*veth pairs*) instead of the switches or LXCs that connect them.

To confirm the validity of this change, we ran three `iperf` [ipe14] clients and one server (concurrently; repeated 100 times) on a topology of thirty nodes (same topology used in the ModelNet comparison) to compare Mininet (using Open vSwitch kernel-based switches) and SNEAC's emulator core (using LXCs as the routers). `iperf` measures the rate at which TCP streams can be sent from multiple `iperf` clients (that generate traffic) to an `iperf` server. Figure 5.2 shows that the performance of Mininet using Open vSwitch switches and the SNEAC core using LXCs is statistically equivalent (using the Kolmogorov-Smirnov test; $p > 0.5$) and that virtual routers can be replaced with LXCs without affecting the validity of the experiment. We note that we do not use Tor for this comparison because vanilla Mininet has no support for running Tor or other applications without custom modifications.

ModelNet and Mininet

Finally, we compare all three emulators together. Figure 5.3 shows the results of an `iperf` test on the SNEAC core, Mininet, and ModelNet, with the same network topology and link characteristics as used in the ModelNet comparison (repeated 100 times). We note two things: the TCP throughput is equivalent for the SNEAC core and Mininet and that they are able to handle the load of concurrent `iperf` clients without affecting the performance. ModelNet again suffers from variable performance, even dropping all packets in some cases.

5.2 Large-Scale Tor Experimentation: Botnets

In August 2013, the Tor network witnessed an unexpected spike in the number of users connecting to the network. As Figure 5.4 shows, the number of clients increased from

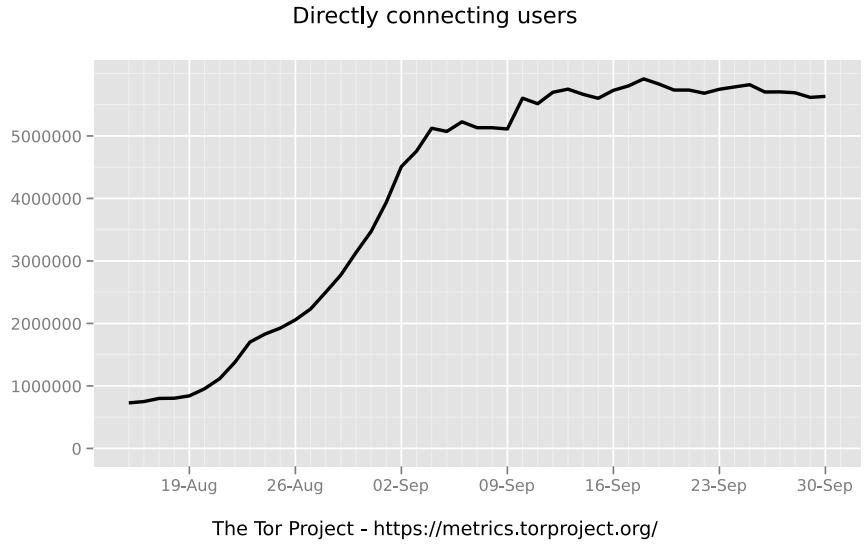

Figure 5.4: Number of users connecting to the Tor network after the botnet attack.

about a million to more than six million in a few days. While early explanations ranged from an increased interest in Tor to political situations in some parts of the world, the sudden and rapid increase was pinned down to the Mevade click-fraud botnet which was running its C&C (command-and-control) server as a Tor Hidden Service [Din13].

Fortunately, the network was able to withstand this attack because the clients were not actually adding traffic to the network, but were instead building millions of circuits, which was draining the resources of the relays since building circuits is an expensive operation involving public-key cryptography. This was further aggravated by two factors: botnet clients were connecting to a hidden service, a process that requires building up to six circuits, and, most of the botnet clients were using an older version of Tor, `tor-0.2.3`, which uses the TAP circuit-level handshake that requires more computational overhead than the current `ntor` handshake.

Because the Mevade botnet was the first known incident of a botnet using the Tor network, it presented an interesting research problem of protecting Tor and its hidden services from botnet abuse. Hopper discussed potential approaches including resource-based and guard-node throttling, reusing failed partial circuits, and hidden service circuit isolation [Hop14], and also evaluated some of these proposals using the Shadow simulator.

For our experiment, we investigate the following problem: consider a botnet that is

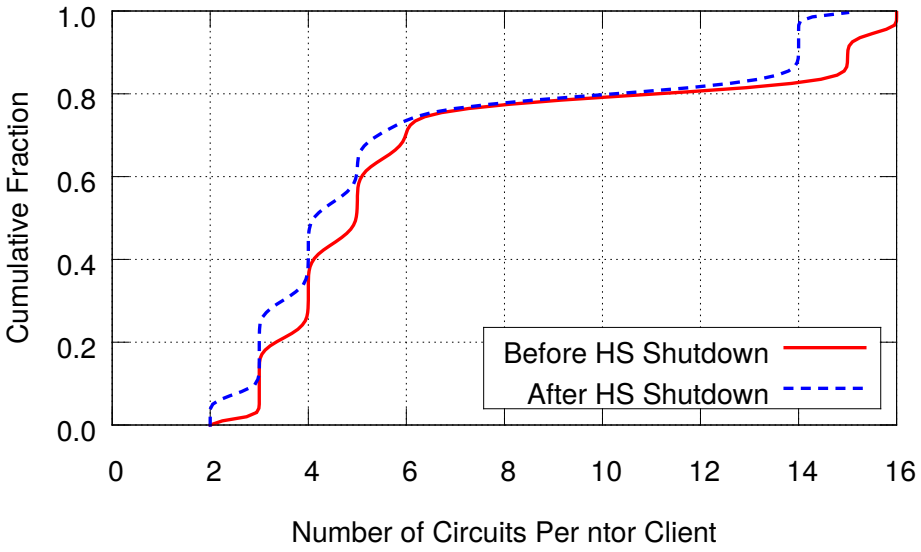

Figure 5.5: The number of Tor circuits per `ntor` client before and after a hidden service is shut down.

using a hidden service as its C&C, with the botnet clients running an older Tor version with the TAP handshake, while the other clients in the network use the `ntor` handshake. We then analyze the network state when the hidden service goes away and the botnet clients start thrashing, trying to reach it.

Our experimental testbed on the CrySP RIPPLE facility consists of five edge nodes connected to a single core machine. The edge nodes are running Tor clients with both TAP (450; representing botnet clients) and `ntor` (2000; representing regular clients) handshakes. Our Tor network consists of three directory authorities, 30 relays, and 2450 clients, spread across five edge node machines. We configure a Tor hidden service on one of the relays to which the botnet clients are connected and periodically fetching data from. The network topology consists of 50 virtual routers (nodes) with 1275 links (edges), with varying node and link characteristics.

We start by taking measurements for the network state in which a hidden service is running and serving TAP clients. To analyze the state of the network, we observe the network characteristics (number of circuits and file download times) for both TAP and `ntor` clients while the hidden service is running. We then turn off the hidden service and as the botnet clients start thrashing the network trying to reach it, we again make a note of the network state. The changes we observe between these two states will help us to

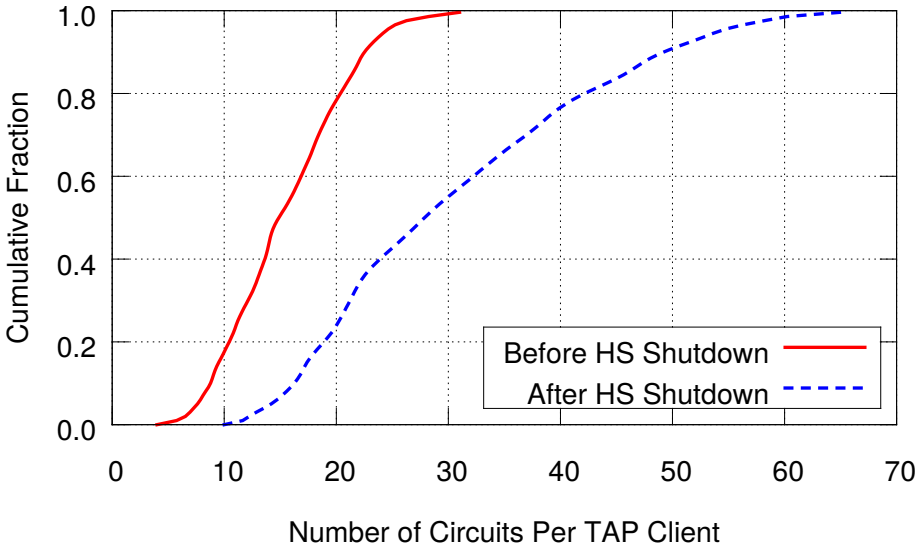

Figure 5.6: The number of Tor circuits per TAP client before and after a hidden service is shut down.

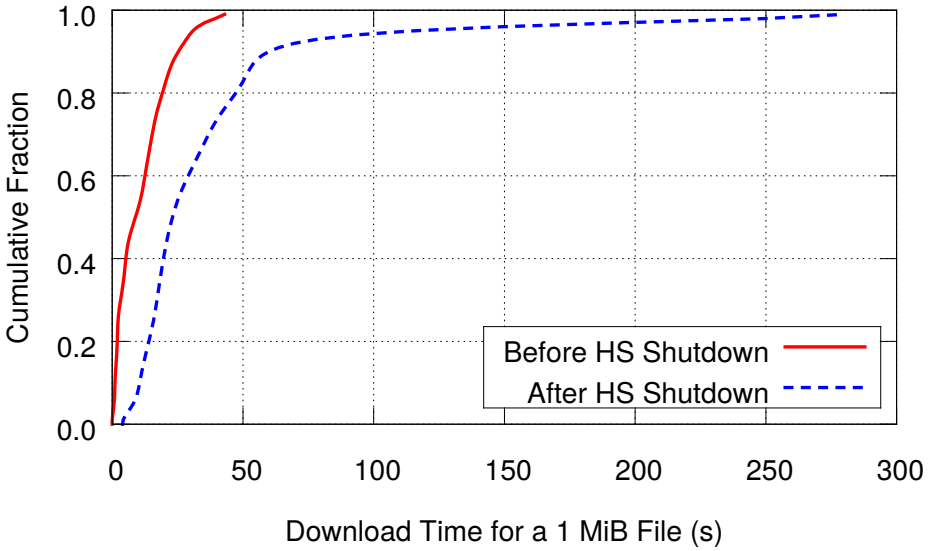

Figure 5.7: The number of seconds it takes to download a 1 MiB file increases after the hidden service is turned off due to an increase in the number of circuits being handled by the relays.

better understand the fluctuation in the Tor network when a busy hidden service in the real world disappears or is shut down.

Figure 5.5 shows the number of circuits per client for 250 randomly sampled `ntor` clients from the network before and after the hidden service is shut down. We note that the number of active circuits of `ntor` clients remains the same. Figure 5.6 shows the number of circuits per TAP client from the network before and after the hidden service is shut down. We notice that the number of circuits *doubles* as the clients start thrashing the network trying to connect to the hidden service which was shut down. This increase in the number of circuits affects the performance of the network, which we measure by the time it takes for a 1 MiB file to be downloaded by the `ntor` clients. Figure 5.7 shows that the number of seconds it takes to download the file *increases* after the hidden service is shut down.

These results confirm our hypothesis that if a hidden service that is acting as a C&C or servicing a large number of TAP clients is taken down, the performance of the network is affected because of the increase in the number of circuits being constructed by the TAP clients as they try to connect to the hidden service. This load is borne by the relays which have to handle more TAP-based circuits after the hidden service shutdown than they were when the hidden service was running. (We recall that TAP clients require more overhead than `ntor` clients.)

Chapter 6

Future Work

To the best of our knowledge, SNEAC is the only emulation-based testbed that allows for the emulation of thousands of Tor nodes, limited only by the physical resources of the experimentation cluster. While a good first step, there are some parts of SNEAC that we will work on improving:

- **Startup Time**: For larger topologies with thousands of nodes and hundreds of thousands of links, setting up and starting an experiment can run into hours because of the time it takes to load and configure the topology. Our threading-based support for Mininet (discussed in Section 3.3.2) decreases the setup time but its Python-based implementation still poses a significant bottleneck. While the time it takes to create LXCs is reasonable (see Table 3.2), creating *veth pairs* and calling `tc` and `netem` can take hours for thousands of links. Because Mininet is just calling standard Linux commands to create the LXCs (a shell process moved into the network namespace) and links (`ip route` and `tc/netem`), it is possible to speed up the process by reimplementing SNEAC in a language like C (which will make it easier to load and manage a topology with hundreds of thousands of links) while at the same time making design decisions that focus on scalability from the very start.
- **Validation for Larger Experiments**: While we have shown that experiments run with SNEAC produce the same results as those with ModelNet (section 5), we would like to validate our emulator for larger topologies with thousands of nodes, comparisons which cannot be made against ModelNet or any other emulator because of their scalability limitations. We should also investigate the scale at which the experimental results get skewed because of lack of CPU resources for the emulation, in which case packets might be dropped and invalidate our results.

Chapter 7

Conclusion

In this thesis, we presented SNEAC, a network emulation testbed for Tor that runs real, unmodified Tor code and can scale to thousands of nodes. SNEAC is based on ModelNet's architecture and uses standard Linux components such as Linux Containers (LXCs), and the `tc` and `netem` modules — this not only helps it to scale better but assures us of the validity of our platform as the emulation is performed by the kernel. As compared to simulators such as Shadow, which also scales to thousands of nodes, SNEAC offers the benefits of emulation: run real Tor code on real operating systems using system libraries, and scalability: leverage all resources of an experimentation cluster by running across multiple machines.

Our implementation is based on an improved version of the Mininet network emulator with many scalability and architectural improvements, and Open vSwitch, a production-quality multilayer virtual switch. The configuration of the virtual routers and network emulation is handled exclusively by the Linux kernel.

We have demonstrated SNEAC's efficacy by comparing it with ModelNet and Mininet, and its scalability by running a large-scale Tor experiment with thousands of nodes to study the effects of a botnet using the Tor network. Even though our focus has been on emulating Tor, SNEAC can be used as a general-purpose network emulator to run any TCP- or UDP-based application, unmodified, on real operating systems in real time. Our emulator is available as open-source software.

References

[BSMG11] Kevin Bauer, Micah Sherr, Damon McCoy, and Dirk Grunwald. ExperimenTor: A Testbed for Safe and Realistic Tor Experimentation. In *Proceedings of the USENIX Workshop on Cyber Security Experimentation and Test (CSET 2011)*, August 2011.

[CCR+03] Brent Chun, David Culler, Timothy Roscoe, Andy Bavier, Larry Peterson, Mike Wawrzoniak, and Mic Bowman. PlanetLab: An Overlay Testbed for Broad-Coverage Services. *ACM SIGCOMM Computer Communication Review*, 2003.

[DET14] DETER. The DETER Project. http://deter-project.org/, 2014. [Online; accessed 10-June-2014].

[Din11] Roger Dingledine. Iran blocks Tor; Tor releases same-day fix. https://blog.torproject.org/blog/iran-blocks-tor-tor-releases-same-day-fix, 2011. [Online; accessed 10-June-2014].

[Din12] Roger Dingledine. Obfsproxy: the next step in the censorship arms race. https://blog.torproject.org/blog/obfsproxy-next-step-censorship-arms-race, 2012. [Online; accessed 10-June-2014].

[Din13] Roger Dingledine. How to handle millions of new Tor clients. https://blog.torproject.org/blog/how-to-handle-millions-new-tor-clients, 2013. [Online; accessed 10-June-2014].

[DMS04] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: The Second-Generation Onion Router. In *Proceedings of the 13th USENIX Security Symposium*, August 2004.

[Emu14] Emulab. Emulab.Net - Emulab - Network Emulation Testbed Home. http://www.emulab.net/, 2014. [Online; accessed 10-June-2014].

[Fre14] The FreeBSD Project. Unsupported FreeBSD Releases. http://www.freebsd.org/security/unsupported.html, 2014. [Online; accessed 10-June-2014].

[GB13] Ian Goldberg and Kevin Bauer. ModelNet Setup. https://cs.uwaterloo.ca/twiki/view/CrySP/ModelNetSetup, 2013. [Online; accessed 10-June-2014].

[Hop14] Nicholas Hopper. Challenges in Protecting Tor Hidden Services from Botnet Abuse. *Financial Cryptography and Data Security*, 2014.

[ipe14] iperf. Iperf - The TCP/UDP Bandwidth Measurement Tool. https://iperf.fr/, 2014. [Online; accessed 10-June-2014].

[JH12] Rob Jansen and Nicholas Hopper. Shadow: Running Tor in a Box for Accurate and Efficient Experimentation. In *Proceedings of the Network and Distributed System Security Symposium - NDSS'12*. Internet Society, February 2012.

[Kar09] Karsten Loesing. Performance of Requests over the Tor Network. 2009.

[KNF+11] Simon Knight, Hung X. Nguyen, Nick Falkner, Rhys Bowden, and Matthew Roughan. The Internet Topology Zoo. *IEEE Journal on Selected Areas in Communications*, October 2011.

[LGA+11] Gilad Lotan, Erhardt Graeff, Mike Ananny, Devin Gaffney, Ian Pearce, and danah boyd. The Revolutions Were Tweeted: Information Flows During the 2011 Tunisian and Egyptian Revolutions. *International Journal of Communication*, 2011.

[LHM10] Bob Lantz, Brandon Heller, and Nick McKeown. A Network in a Laptop: Rapid Prototyping for Software-Defined Networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, page 19. ACM, 2010.

[Lin09] Linux Foundation. netem — The Linux Foundation. http://www.linuxfoundation.org/collaborate/workgroups/networking/netem, 2009. [Online; accessed 10-June-2014].

[LXC14] LXC. LXC - Linux Containers. https://linuxcontainers.org/, 2014. [Online; accessed 10-June-2014].

[MAB+08] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. OpenFlow: Enabling Innovation in Campus Networks. *ACM SIGCOMM Computer Communication Review*, 2008.

[Mod05] ModelNet. ModelNet Release. http://modelnet.ucsd.edu/release.html, 2005. [Online; accessed 10-June-2014].

[ns-14] ns-3. The ns-3 Network Simulator. http://www.nsnam.org/, 2014. [Online; accessed 10-June-2014].

[Ope14] Open vSwitch. ovs-vswitchd. http://openvswitch.org/cgi-bin/ovsman.cgi?page=vswitchd%2Fovs-vswitchd.8, 2014. [Online; accessed 10-June-2014].

[PPA+09] Ben Pfaff, Justin Pettit, Keith Amidon, Martin Casado, Teemu Koponen, and Scott Shenker. Extending Networking into the Virtualization Layer. In *HotNets*, 2009.

[Pro11] The Tor Project. ntor-handshake. https://gitweb.torproject.org/torspec.git/blob/HEAD:/proposals/216-ntor-handshake.txt, 2011. [Online; accessed 10-June-2014].

[Riz97] Luigi Rizzo. Dummynet: A simple Approach to the Evaluation of Network Protocols. *ACM SIGCOMM Computer Communication Review*, 1997.

[SGR97] Paul F. Syverson, David M. Goldschlag, and Michael G. Reed. Anonymous Connections and Onion Routing. In *IEEE Symposium on Security and Privacy*, 1997.

[Sha14a] Shadow. Pull Request #197 from jdgeddes/master. https://github.com/shadow/shadow/commit/613778d64bc9ebcc93d2a61aa37a803a20cf47da, 2014. [Online; accessed 10-June-2014].

[Sha14b] Shadow. Using the Scallion Plugin. https://github.com/shadow/shadow/wiki/Using-the-scallion-plug-in, 2014. [Online; accessed 10-June-2014].

[SPBP06] Neil Spring, Larry Peterson, Andy Bavier, and Vivek Pai. Using PlanetLab for Network Research: Myths, Realities, and Best Practices. *SIGOPS Operating Systems Review*, 2006.

[Tor13] The Tor Project. Tor Research. https://research.torproject.org/, 2013. [Online; accessed 10-June-2014].

[Tor14a] The Tor Project. Chutney. https://git.torproject.org/chutney.git, 2014. [Online; accessed 10-June-2014].

[Tor14b] The Tor Project. TC: A Tor Control Protocol. https://gitweb.torproject.org/torspec.git?a=blob_plain;hb=HEAD;f=control-spec.txt, 2014. [Online; accessed 10-June-2014].

[VYW+02] Amin Vahdat, Ken Yocum, Kevin Walsh, Priya Mahadevan, Dejan Kostić, Jeff Chase, and David Becker. Scalability and Accuracy in a Large-Scale Network Emulator. *ACM SIGOPS Operating Systems Review*, 2002.

[Wat14] Waterloo, University of. CrySP RIPPLE Facility. https://ripple.uwaterloo.ca/, 2014. [Online; accessed 10-June-2014].