TU WIEN Informatics

At One's Own Pace

"Walking Onions" and resource-constrained devices in the Tor network

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Software & Information Engineering

by

Ivaylo Ivanov
Registration Number 11777707

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.-Prof. Dipl.-Ing. Mag. Dr.techn. Edgar Weippl
Assistance: Dipl.-Ing. Dr.techn. Wilfried Mayer

Vienna, 17th March, 2022

Ivaylo Ivanov	Edgar Weippl

Erklärung zur Verfassung der Arbeit

Ivaylo Ivanov

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 17. März 2022

Ivaylo Ivanov

Abstract

Walking Onions is a set of protocols, aimed at bringing performance improvements to Tor with regard to circuit construction. The original research includes a simulation and provides an empirical proof that the amount of bandwidth used for connecting to the onion network has been greatly reduced.

The aforementioned simulation, however, has been performed in a rather static manner and a usability assessment from the client point of view is missing. In this thesis, we conduct the same simulations with dynamic and publicly available bandwidth sample sets from 2019 and 2021. Additionally, we also define two theoretical scenarios - one where all fast relays (with bandwidths above the upper quartile in the sample set) suddenly go offline and one where all slow relays (with bandwidths below the upper quartile) suddenly go offline, and perform the simulations for them as well. Finally, we perform usability assessment from the client point of view by calculating the time required to construct a circuit and approximating it to real-world data from 2019 and 2021.

The results show that the total bandwidth required for circuit construction during an epoch is a negligible part of the total throughput of the network for that same period. A trade-off for this improvement is also presented - because of the additional complexity of the protocol stack, the time required to construct a circuit will be larger than the original Tor implementation.

Contents

CHAPTER 1

Introduction

Since its initial release, Tor has played an important part of the daily lives of a lot of people - from whistleblowers and citizens of oppressive states to normal people who want to keep their privacy online. According to the official Tor user metrics [1] for 2021, Tor relays have seen a constant usage of about 2 million people worldwide.

Despite its popularity, Tor, as any software, is not perfect. One of its most crucial issues that need to be addressed is speed. As the "Frequently Asked Questions" section of their website [2] suggests, Tor's less than ideal speed is a result of the inherent design of the anonymity network. This issue, however, prevents users from using the Tor browser as their only everyday browser and puts it in a category of a companion browser. Such behavior from the users is dangerous as inconsistent usage may lead to de-anonymization [3].

Recently, a lot of research has been focused on improving Tor's speed and performance. One such advancement is the work of Komlo et al. [4], called *Walking Onions*, which introduces a new set of protocols that aims to make the anonymity network more scalable and to reduce the metadata that needs to be passed for circuit construction while retaining the initial anonymity of the users.

In this thesis, we will take a look into how the different *Walking Onions* protocols perform on resource-constrained devices by evaluating different theoretical scenarios using a reworked version of simulator, provided by the authors of the paper [5]:

- will the set of protocols offer the same speed improvements when taking into consideration the bridge and relay landscape at the time of writing;
- what will happen if the "fastest" Tor relays at the time of writing suddenly became offline and only the slower ones were left;

- what will happen if the opposite situation occurs - the slower relays are removed and only the fastest ones are left?

Additionally, we will investigate if there are any trade-offs in using *Walking Onions* and if these potential negatives outweigh the positives.

The analysis is split in different chapters. Background work is presented in chapter 2. Related work can be found in chapter 3. The methodology of the study and the results are presented in chapters 4 and 5 respectively. The results are then discussed in chapter 6. The work is concluded in chapter 7.

CHAPTER 2

Background

In this chapter, we will review how connections, or *circuits*, are constructed in the traditional ("vanilla") Tor protocol [6], what components are needed and how the proposed improvements compare.

2.1 Relays and directory servers

The Tor network is based on community-run Onion Routers or *relays* [6]. There are approximately 6500 of them at the time of writing [7]. Each relay maintains a long-term identity key, which is used to sign the relay's own *relay descriptor* [8]. Thus, it may later be verified that the relay descriptor comes from its expected source, assuming a non-compromised key. Apart from the numerous other details, the descriptor contains the router's IP address, bandwidth and the router's *onion key* - a short-term key that is used by the client to construct a circuit using this relay.

Vanilla Tor circuits normally contain three types of relays, separated in different layers (hence, onion routing). The relays can be categorized in the following manner:

- guard relay - the guard relay is the closest to the client and the one that receives the initial connection request;
- exit relay - the exit relay lies the farthest from the client and is the one that forwards the traffic directly to the destination;
- middle relays - one or more relays that lie between the guard and the exit. They are responsible for forwarding the traffic securely to the next hop.

According to their position in the circuit, each relay knows a subset of the whole information the client knows:

- the guard relay knows the source, but not the destination of the traffic;
- the middle relays know neither the source, nor the destination of the traffic;
- the exit relay knows the destination, but not the source of the traffic.

Additionally, each relay has a different key for encrypting the communication to the next relay [6]. Thus, perfect forward secrecy for the data is preserved.

Apart from relays and clients, there is also another type of nodes in the Tor network - *directory servers* [6]. They are highly available entities that are responsible for gathering relay information through the signed relay descriptors and presenting it to the clients and other relays. Each hour (this interval is referred to as "epoch" later on), the directory servers evaluate the gathered relay data, vote on it and produce a signed document called a *consensus document* [8], containing all relays that are suitable for routing traffic through the Tor network. The clients and other relays then fetch and cache the data and use it to build circuits.

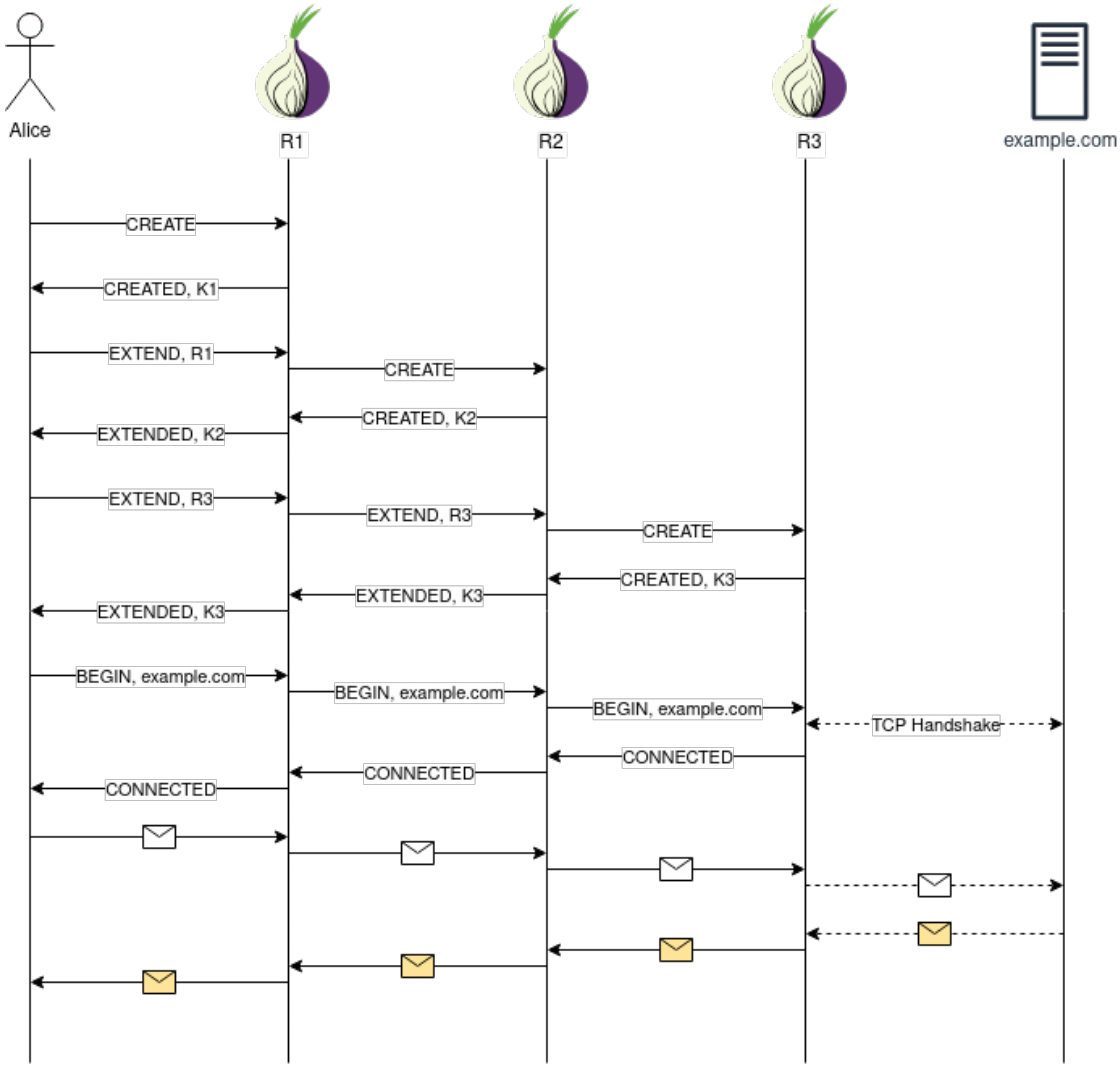
2.2 Vanilla Tor protocol - circuit building

Vanilla Tor circuits are constructed in a telescoping manner (for simplicity, the format of the requests or "cells" has been omitted):

1. the client sends a CREATE request to the guard relay of the chosen path and both negotiate a session key for the communication;
2. the client sends an EXTEND request, containing an identifier of the next hop, to the guard relay. The guard relay may decapsulate the request and create a new CREATE request to the next hop or simply forward it to an already existing next hop. When the connection has been established the same way as the previous step, an acknowledgement, containing the new session key, is sent to the client. This step is repeated as much as needed;
3. when the connection to an exit relay has been established, the client may continue with normal TCP traffic that gets forwarded through the circuit. The traffic between the exit relay and the destination may be running in clear text if the destination is not using encryption or is not running as a Tor hidden service [6].

The process can be seen in Figure 2.1 - the dotted line represents (potentially) plaintext traffic, whereas the solid represents encrypted traffic.

Figure 2.1: Vanilla Tor circuit construction



2.3 *Walking Onions*

2.3.1 Overview

Walking Onions refers to a set of protocols, described by Komlo et al. [4], which aims to create a new way for circuit extension in Tor that should make the network drastically more scalable and reduce the amount of metadata that the Tor network participants need to possess and process in order to construct a circuit.

In this section, we will first introduce the new data structures that *Walking Onions* proposes and compare them to the ones aforementioned in section 2.1. Afterwards, we will briefly cover two different algorithms for authenticating the new data structures from clients and relays. Finally, we will summarize the two methods for circuit construction from the paper and compare them to the "vanilla" one.

2.3.2 SNIPs and ENDIVEs

As with the vanilla implementation of Tor [6], the *Walking Onions* set of protocols also feature a document, created by directory servers on a per-epoch basis - the *Efficient Network Directory with Individually Verifiable Entries*, or ENDIVE. Each ENDIVE entry is called a SNIP - *Separable Network Index Proof*.

ENDIVEs are fetched by relays when bootstrapping and updated by obtaining only the changes in regular intervals (once per epoch). With *Walking Onions*, clients do not require the whole document to construct a circuit.

Each SNIP serves the same function as a relay entry in the vanilla consensus document. They contain the same information about relays as the vanilla Tor relay descriptors [8], but they expand it with three more important fields:

- *index range* - a range of integers showing how probable it is for this relay to be included in a circuit;
- *own authentication tag* - each SNIP is signed by the directory servers only over its content. As such, clients can fetch individual SNIPs and not the whole ENDIVE;
- *timestamps* - two timestamps indicate the creation and expiration of a SNIP.

Both ENDIVE and SNIPs are authenticated by the directory servers so that relays and clients know that they can trust them. A short overview of how this is achieved and what algorithms are used is available in the next subsection.

2.3.3 Authentication algorithms

Walking Onions can authenticate the previously introduced data structures - SNIPs and ENDIVEs, in three ways.

The most naive implementation is to have one signature per voter - each directory signs the ENDIVE and the SNIPs. This way, the ENDIVE contains $N_d \cdot N_r$ signatures, where N_d is the number of directory servers and N_r - the number of relays. This is not ideal, as the relays need to fetch more information when bootstrapping.

The second way is to use joint signatures - a single signature represents an arbitrary number of signers. This means that the directory servers need to coordinate to sign the entity. In the case of *Walking Onions*, threshold signatures [9] have been used - only a subset of all signers need to sign.

The final way proposed is using Merkle trees [10]. The client can verify that a specific SNIP is within the ENDIVE, signed by the directory authorities by downloading the root of the Merkle tree from the ENDIVE. Later on, this root is used in conjunction with a proof, saved in the SNIP that the client downloads. This proof should show that the SNIP can be obtained by following a path from the previously acquired root, thus proving its validity.

Regardless of the way the directory servers sign the document, the clients and relays only need to verify the authentication tag attached to the SNIP or the ENDIVE.

2.3.4 *Walking Onions* - circuit building

The *Walking Onions* set of protocols have two variants of circuit construction. The first one is in a telescoping manner, similar to the vanilla Tor approach. The second one aims to create the circuit in a single pass - the client contacts the guard relay and it handles the rest of the extension, delivering all the keys to the client in a bulk at the end.

In this subsection, we will look into both protocols and describe their advantages and disadvantages compared to vanilla Tor.

Telescoping *Walking Onions*

As mentioned previously, telescoping *Walking Onions* works similarly as the vanilla implementation, described in section 2.2 and displayed in Figure 2.1.

There are a couple of differences, the first being that the client does not select the next hop manually, but instead generates a random index between zero and an arbitrary value α (the original paper recommends the value of 2^{32}). This value gets sent with the extension request to the last hop of the already established circuit (for example, R_n). The relay R_n then fetches the SNIP from the consensus that has the chosen index in its index range. This will be the next hop of the circuit (R_{n+1}). The confirmation messages are also adjusted to include the SNIP of the chosen relay R_{n+1} alongside with the key. The client has the signed network parameters and can verify the SNIP of R_{n+1}. Furthermore, the client verifies that R_{n+1} has been chosen fairly by checking if the selected index is really within the range, specified in the SNIP.

Clearly, this protocol presents minimal changes that need to be done to the vanilla implementation to work. This is not the case with the next type of the protocol.

Single-Pass *Walking Onions*

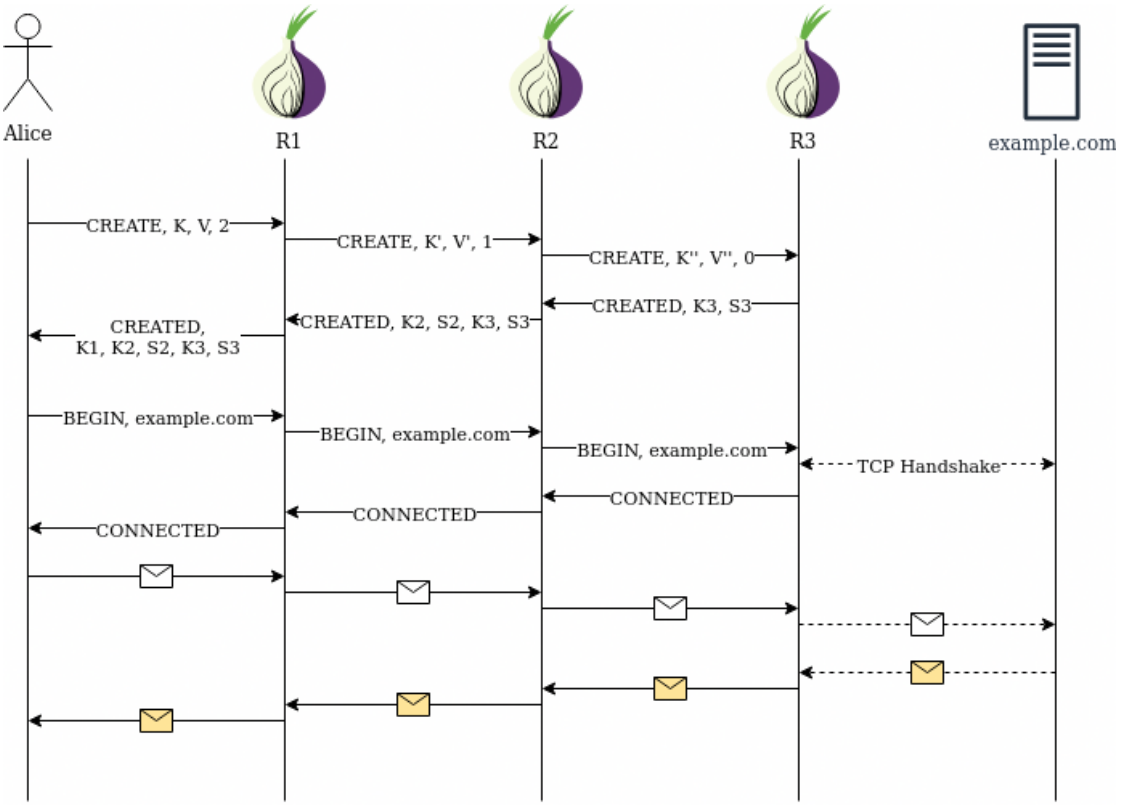
Single-Pass *Walking Onions* is a newly-proposed path selection algorithm, which, unlike the telescoping variation, aims to build the whole circuit with a linear number of messages, relative to the number of relays in the circuit. This has a positive effect on the bandwidth, available for the Tor network, as the amount used for circuit construction has been greatly reduced.

Circuit construction with the protocol continues roughly as follows:

1. the client generates two ephemeral Diffie-Hellman key pairs - one for deriving session related keys and one for performing index calculation. Additionally, an integer TTL (time to live) counter is also created. This counter shows the number of hops that the circuit should have and indicates to the last circuit to terminate further construction. The public keys and the TTL counter are sent to the guard relay;
2. the next hop in the circuit receives the keys and the TTL counter;
3. the relay uses its own set of keys and the ones from the client to generate a shared secret for the circuit expansion;
4. a random index is generated. This is not a hard requirement solely for the client like in the telescoping protocol, but rather a shared responsibility between the client and the relays. The index is also created in such a way that the client can verify that no interference by a third party has occurred. This random index is then used for selecting the next relay in the same matter as in the telescoping variant;
5. the current relay signs the keys from the client with its own private keys, reduces the TTL counter by one and forwards the signed keys and the TTL to the next relay;
6. if the next relay is not the final relay of the circuit, it repeats the same actions;
7. the final relay responds with its own session key and SNIP, which then both get forwarded down the chain until they reach the client;
8. the client verifies the SNIPs. At this point in time, the client has also gathered all the relays' keys;
9. the communication with the target proceeds as normal.

A graphical representation of the protocol can be seen in Figure 2.2. It is important to note that this is a simplified explanation of the process. The exact algorithm is out of the scope of the thesis, but an interested reader will find more information in the original research paper.

Figure 2.2: *Walking Onions* - Single Pass circuit construction



CHAPTER 3

Related work

In this chapter, we will give an overview of the work that inspired or is connected to this thesis. Section 3.1 will present different ways to model and evaluate Tor performance. In section 3.2 state-of-the art solutions for Tor experimentation will be described.

3.1 Tor performance evaluation

Performance has always been a pain-point for Tor [2] and is a really active field of research for potential improvements. Because of this, there are numerous ways to evaluate performance. All of them, regardless of their count, need a metric that acts as a model for the analysis. Picking a suitable metric before evaluating performance data is called *performance modelling*. Snader and Borisov [11] argue that one metric that can be used as a performance model for evaluations is the node throughput, which is directly linked to its bandwidth. As bandwidth already plays a large enough role in the consensus creation process, this statement is not unreasonable. An additional proposal by Murdoch and Watson [12] state that another such metric is the expected processing time for a cell. In addition, they also argue that the bandwidth is usable as a measure for usability assessment, but does not account for the effect of modifying the whole network (for example, all nodes adopt a new circuit-building protocol or path selection algorithm).

As the ultimate goal is to improve the overall experience for the client, performance evaluation should be done alongside *usability assessments*. The paper by Müller et al. [13] describes a usability assessment experiment that uses latency for HTTP requests experienced by the client as a metric for Tor usability. This is also not unreasonable, as high latency may drive new users away and thus hinder the growth of the Tor network. The experiment is conducted with clients, located on different continents, that perform requests to the websites in the SEOmoz Top 500 list [14]. The latencies are then gathered and evaluated against each other. Additionally, an analysis of the user tolerance has also been conducted.

3.2 Tor experimentation

Generally, Tor experimentation aims to find the balance between realism and safety - the evaluations need to be as realistic as possible without providing any real dangers to the live Tor network. There are many existing evaluation methods - from abstract models of the Tor network, through network simulators up until running tests "in the wild". Shirazi et al. [15] categorized them in six categories:

- analytical/theoretical modeling;
- private Tor networks;
- distributed overlay network deployments;
- network simulation;
- network emulation;
- live Tor research.

Out of all six, we are only going to focus on network simulation and network emulation, as they are relevant for this thesis.

ExperimenTor [16] is a whole-Tor network testbed, aimed at performing *Tor network emulation*. The tool relies on two components for its functionality - clients, running the Tor source code directly, and a ModelNet [17] network emulator that connects the clients in a network. The clients normally run in a virtualized environment and the emulator connects them to a virtualized network topology. One advantage to this approach is that Tor performance improvements can be directly tested in a controlled environment running an improved version of the software.

Shadow [18] is a plugin-based *network simulation platform* that allows running Tor experiments on a single machine. The Tor plugin allows the user to run real Tor source code and perform experiments in a flexible and reproducible manner. This method has the same advantage as ExperimenTor - namely, researchers can test with the real codebase and submit improvements quickly to the upstream. One could argue that it is even more flexible than ExperimenTor, as the users are not required to have a cluster at their disposal - only a single machine with enough resources will work just as fine.

Both of these models are not perfect. They have two issues that need addressing: sampling errors may occur when running simulations and they do not suit all needs as radical changes normally need to be prototyped first.

Jansen et al. [19] address the first issue by grounding the foundations for *sound statistical interference* for Tor networks. It is argued that because most simulations are sampled, a sampling error may occur, skew the results and cause Type-I or Type-II errors in the conclusion. The researchers have also incorporated their findings in Shadow.

The second issue is solvable by developing a custom simulation that focuses only on the problem at hand before contributing to the Tor codebase. As such, Komlo et al. have created a simulator[5] just for the *Walking Onions* stack of protocols.

3.3 Summary

By recapitulating the aforementioned works and viewing them in the context of *Walking Onions*, it is visible that an experiment for evaluating the proposed protocol design is needed.

The experiment must be as realistic as possible without going out of the scope of the thesis - if no actual implementation is available for emulation, then simulation should be considered. As the new protocol stack proposes improvements for the user, a suitable user-focused metric should be chosen for performance modelling. Additionally, the simulator should allow usability assessment by simulating multiple scenarios from the client perspective using a model that takes into account the fact that the whole network has undergone a drastic change. Finally, the simulator should, ideally, take statistical interference in account when gathering the results.

The original simulator already covers the first two requirements. The third requirement has not been covered in the original paper and is out of the scope of the thesis.

In the next chapter, this work will introduce the following improvements to the experiment:

- enhanced realism - make it possible to supply real-life data;
- additional scenarios - use a similar performance metric to the one by Snader and Borisov [11] to define different performance classes and evaluate the stack with them;
- usability testing - using a similar performance metric to the one by Murdoch and Watson [12], perform a usability assessment from the client perspective.

CHAPTER 4

Methodology

In this chapter, the methodology for evaluating different scenarios for the *Walking Onions* set of protocols will be presented. In section 4.1, different simulation scenarios will be presented. In section 4.2, the evaluation of circuit building times will be described. In section 4.3, the changes and functionality that were added to the simulator will be presented. In section 4.4, the initial attempts for running simulations will be shortly covered.

4.1 Simulation scenarios

Because of its design [6], the Tor network relies on volunteers to function - relays run on diverse hardware, in different parts of the world, with different people operating them. This distribution of the network resources is good for several reasons: anonymity is improved and there is no central entity in control of the network. Unfortunately, it also means that sometimes a Tor relay is ran on inadequate hardware - either from the perspective of system resources or available bandwidth. There are ways to tackle the issue - directory servers tend to favor faster nodes over slower ones, but there is no guarantee that your circuit will not get a bottleneck by going over an underpowered hop. Another solution would be simply not including the slow nodes in the consensus. When only a handful of nodes are used, however, anonymity will suffer as paths will become predictable and entities, operating a large enough amount of nodes, may be able to perform correlation attacks.

This raises the question if *Walking Onions* could solve the issues with technically inadequate devices. To answer this, we decided to look into how *Walking Onions* would perform under different theoretical scenarios. These scenarios are presented in the following paragraphs.

4.1.1 Current network landscape

The first question that arises is what improvements does *Walking Onions* bring in the current Tor network landscape? The original research already examined this, and even though the evaluation was based on measurements from 2019 (see section 4.3), it provided a solid basis that was expanded and improved to include more recent data. We ran simulations using recent data about the whole network, including bridges.

4.1.2 "Running Onions"

As previously mentioned, Tor is run on a number of inadequate devices. However, there are also quite a few organizations that are dedicated to running Tor relays only, called Relay Associations [20]. Normally, these nodes are reserved only for this usage and provide lots of bandwidth to the network. This brings us to the following scenario: what would the impact on the network performance (not anonymity) be if we left Tor running only on performant nodes? The idea is to discover whether *Walking Onions* really brings any meaningful improvements in this scenario over running the vanilla implementation.

4.1.3 "Strolling Onions"

Naturally, we also wanted to consider the opposite: what would happen if only the "slower" nodes were left? This scenario is more interesting. It aims to evaluate whether *Walking Onions* would be able to bring the performance of such a Tor network on par with the current one. It also has another implication - if we observe a large improvement in the available bandwidth, this would mean that technically inadequate (by the current standards) relays may be able to handle more traffic and be used in more circuits, thus improving anonymity.

4.1.4 Defining slow and fast nodes

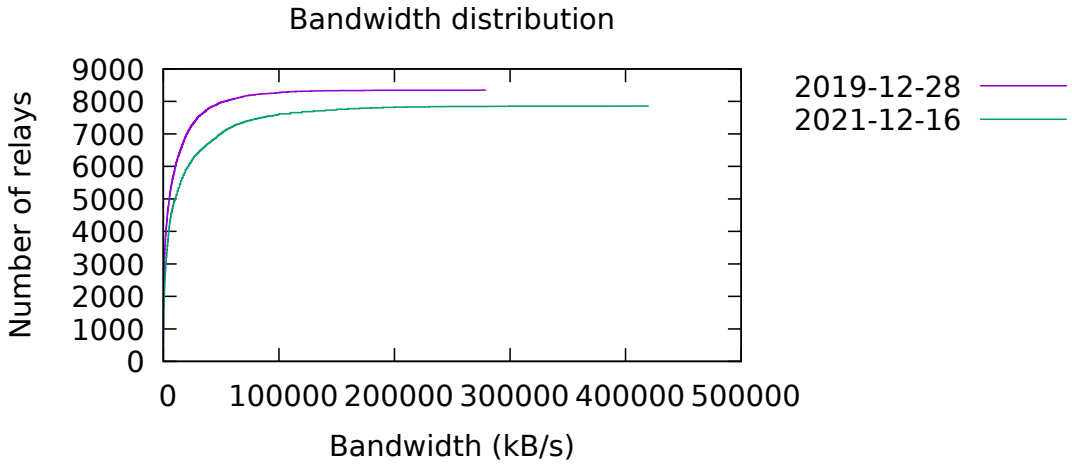
In order to do this separation, we need to define what "fast" and "slow" nodes mean. After analyzing our datasets, the conclusion was reached that we will refer as "fast" to the nodes that have bandwidths above the upper quartile of the set of relay bandwidths, and as "slow" - to the rest. A graph supporting the split is available in Figure 4.1. It is based on the datasets used in the simulations.

All the scenarios have been simulated on the 5% network scale of 2019 and 2021 sample data. The results can be seen in section 5.1.

4.2 Circuit building times

The *Walking Onions* set of protocols propose new ways of building circuits, one of which is similar but the other drastically different from the vanilla implementation. As such, an important aspect to evaluate is the speed at which a circuit can be built. When calculating circuit build times, we need to give answers to two questions: how are the

Figure 4.1: Bandwidth distributions, 2019 & 2021



measurements conducted in the simulator and how do we evaluate the results with regard to real-world performance.

4.2.1 Conducting measurements

The measurements are conducted directly in the simulator by pinpointing the exact time when the circuit construction started and subtracting the result from the time when the circuit construction finished. Because the speed of the simulator is based on the current clock speed of the Central Processing Unit (CPU) of the machine and does not take into account real-world interferences like network bottlenecks and physical distance, the end result is returned as a decimal number, representing fractional seconds. These results by themselves are useless, but they allow us to calculate a value to which we are going to refer to as *circuit building ratios*.

4.2.2 Circuit building ratios

In sections 2.2 and 2.3, we presented how circuits under different protocols are constructed. The vanilla implementation of the construction algorithm is the most resource-friendly in terms of computational power as it does not rely on performing so many calculations but rather on parsing already-available data. As such, we can make the assumption that the simulator will construct circuits the fastest when performing vanilla simulations. This means that we can take each *Walking Onions* measurements and divide them with the vanilla ones to get the ratio in which they are related. The calculated ratios should remain unchanged through different simulations, regardless of the power of the underlying system, as the simulator always completely monopolizes a whole core throughout an epoch.

4.2.3 Ratios and real-world speeds

The speed of circuit construction is relative to the different Tor users and can be impacted by multiple factors - from slow connection to low-powered CPU. In order to do an evaluation of circuit building speeds, one needs a dataset that is representative of the different factors. At the time of writing, such a dataset is offered by the Tor metrics project [21]. In order to obtain a basis for vanilla circuit construction to use with the ratios from the previous paragraph, we do the following:

- parse the file, according to the current specification [22];
- sum the median circuit building times for all three positions in a region;
- calculate the average times, using the sum above.

The end result is then multiplied by each of the ratios from the previous paragraph. Thus, we obtain a speed that can give us insight in how the protocols would perform when constructing a circuit in real-life. The code for performing the calculation is available in algorithm 4.1.

Algorithm 4.1: Relative circuit building time calculation

Input: list of dicts of real-world circuit building times C for source S, dict of local circuit building times L
Output: dict of relative times T for source S

```
1  s ← 0;
2  l ← len(C);              /* note:  l is always 3 per the spec */
3  for i ← 0 to l − 1 do
4  |  s ← s + C[i]["md"];              /* s holds the median sum */
5  end
6  b ← s/l;                /* b holds the average of the medians */
7  R ← {};                     /* R holds the ratios as in 4.2.2 */
8  v ← L["vanilla"];
9  for k in L.keys() do
10 |  R[k] ← L[k]/v ;             /* k represents a protocol type */
11 end
12 T ← {};
13 for k in R.keys() do
14 |  T[k] ← R[k] ∗ b;
15 end
```

The relative circuit building times have been calculated on the 5% network scale of 2019 and 2021 sample data. The results can be seen in section 5.3.

4.3 Simulator changes

At the time of writing, the *Walking Onions* simulator makes the following assumptions when running the simulations: the relay landscape is approximated to the one from December 2019 and the number of relays is statically set to 6500. Both are visible in the source code of the simulator [23] [24].

The assumptions from above make it hard to reevaluate the stack using a newer set of data. In order to do so, one needs to change the statically set bandwidth distribution algorithm with one that produces an approximation for a more recent point in time. Additionally, one must also change the statically set number of relays.

To tackle this issue, the following approach has been taken:

- an automatically exported Tor bandwidth metrics file [25] gets fed to the simulator;
- the Simple Bandwidth Scanner [26] (sbws) bandwidth files get parsed by the simulator, following the specification [27]. Bridges are also included, relays with `bw=1` get discarded as they are not included in the consensus;
- the simulator automatically counts the elements in the parsed data from the previous step. Each element represents a relay or a bridge. Thus, the number of relays is obtained dynamically;
- the parsed data, the sample size and the total number of relays get submitted to an adjusted version of the sampling algorithm, presented by Jansen et al. [28], which produces a bandwidth distribution for the simulation. The adjustments are detailed in algorithm 4.2 and they aim to produce the same number of results as the sample size. The algorithm returns a distribution that is skewed towards the slower relays to limit the interference. Due to time constraints additional improvements have not been implemented.

With this approach, it is now possible for the user to feed the simulator real-world samples, which get dynamically parsed and evaluated. The original algorithm by Komlo et al. has been kept as a separate function for fallback. One drawback of this approach is that the simulator only supports sample sizes of at most 50 percent of total relay count in the bandwidth file supplied. This, however, does not hinder our simulation.

Additionally, in order to simulate the scenarios in section 4.1, a bandwidth file generator was needed. The generator parses the sbws bandwidth files, sorts the data and splits it in quartiles. A command line argument specifies whether the output file only needs to include relays with bandwidth above or under the upper quartile as per section 4.1. The produced files can be directly fed to the simulator.

The easiest way to evaluate bandwidth differences between years is with a plot. This feature has been added to the simulator along with the dynamic bandwidth distribution

Algorithm 4.2: Adjusted Jansen distribution algorithm

Input: sorted list L of N relay bandwidths, sample size K
Output: sorted list of sampled bandwidths S of size K

$n \leftarrow floor(\frac{N}{K})$;
$r \leftarrow K - n$;
$i \leftarrow 0$;
for $k \leftarrow 0$ *to* $K - 1$ **do**
    $j \leftarrow i + n$;
    if $k < r$ **then**
        $j \leftarrow j + 1$;
    end
    $bin \leftarrow L.slice(i, j)$; `// range [i, j)`
    $S.add(median(bin))$;
    $i \leftarrow j$;
end
$l \leftarrow len(S)$;
if $l < K$ **then**
    for $c \leftarrow 0$ *to* $K - l$ **do**
        $S.add(S[c])$; `// append from beginning until |S| = K`
    end
end

calculation. The sbws bandwidth files get parsed as mentioned previously, sorted and then plotted.

The evaluation for circuit building times highlighted in section 4.2 has also been implemented in the simulator.

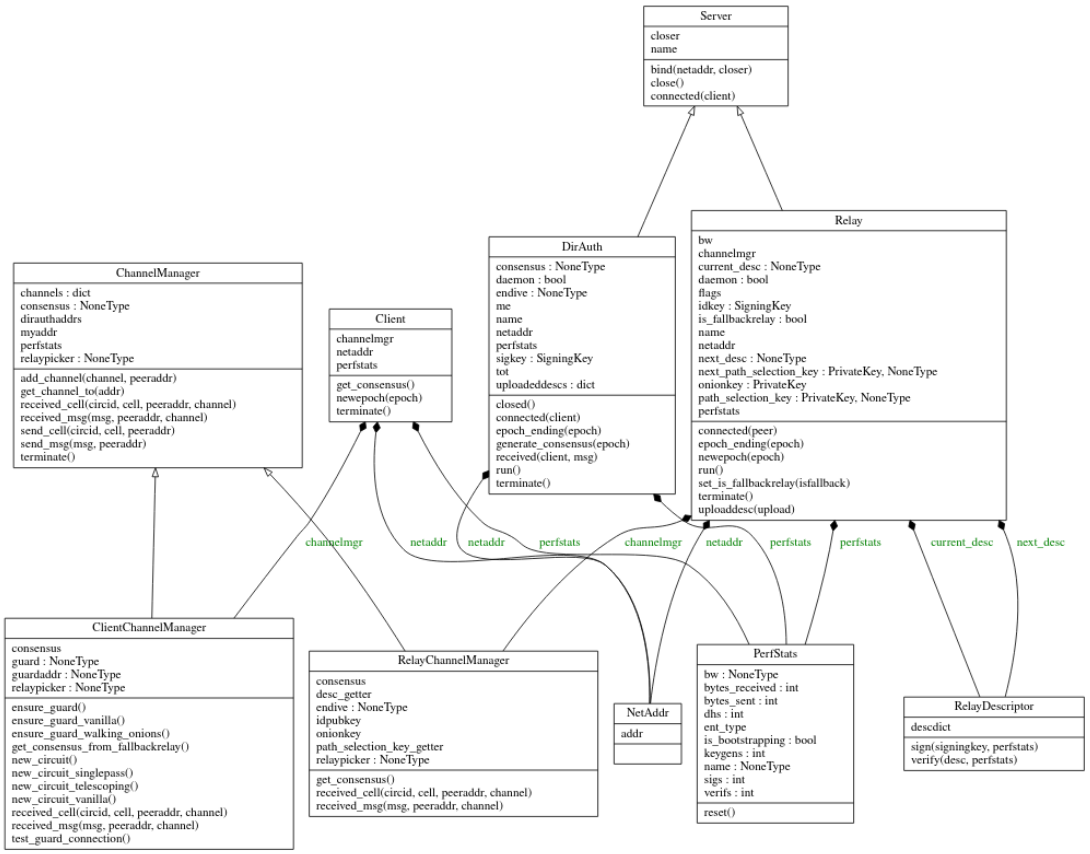
4.4 Initial attempts

There were two initial ideas for simulating *Walking Onions* on resource-constrained devices - creating a real-world simulation with machines, running the *Walking Onions* stack implementation, and adapting the simulator by Komlo et al. to have simulated resource-constraints.

The first idea was immediately discarded as an official implementation for the protocols had not yet begun at the time of writing. The second attempt included splitting the simulator in different processes, which in turn can be made to simulate memory and processing time limitations by using the multiprocessing Python library [29].

The simulator implementation is split into classes. In order to make the different parts run in parallel, we are interested mainly in `Server`, `Client`, `Relay`, `DirAuth` (UML diagram can be seen in Figure 4.2).

Figure 4.2: Simulator UML diagram (relevant classes)



As visible from the diagram, the `DirAuth` and `Relay` classes both extend the `Server` class. As such, the idea for the implementation was to make the `Server` class extend Python's `Process` class [30] and let each subclass implement the `run()` method as per the needed functionality. Then, each process would get artificially limited using Python's resource library [31], which would simulate a resource-constrained device.

After implementing the functionality from above, the relays and the directory servers were able to start as different processes and they were getting their limits set. However, the simulator is constructed in such a way that the most work gets done in a single process regardless of whether the servers are running in different processes. This resulted in one main process and dozens of child-processes that were running and staying in a busy-waiting state until the parent was finished. Additional inspection showed that the whole communication system would also need to be re-implemented in a process-safe manner, which would mean rewriting the entire simulator from scratch. This was out of the scope of the thesis and the second idea was also discarded.

CHAPTER 5

Results

In this chapter, the results from the tests, described in chapter 4 are presented. In section 5.1, the results from the different simulation scenarios are presented. In section 5.3, the results from the circuit building times tests are presented. All the code, analysis scripts, datasets and results are available at `https://git.ivayloivanov.eu/ivo/walkingonions-boosted`.

5.1 Simulation scenarios

As mentioned in section 4.1, the results below have been obtained by simulating the different scenarios on the 5% network scale of 2019 and 2021 sample data.

We will analyze three different metrics: total relay bytes - per client and per relay, and total client bytes.

The first metric is total relay bytes, which measures the total number of bytes a relay sends or receives per epoch. An important remark is that the data in the *Walking Onions* paper have been normalized, meaning that the total accumulated bandwidth has been divided by the number of relays. This is not a problem, because the formula for calculating the number of relays (N) only contains one variable: $N = S * 6500$, where S represents the network scale and $0 \leq S \leq 1$. In our case, however, the number of relays differs from scenario to scenario. The formula $N = S * T$ replaces the constant with an additional variable T, which denotes the total number of relays in the given scenario (the size of the distribution, returned by algorithm 4.2). Thus, in order to draw comparisons between the results here and the ones obtained in the original research, we are going to use an alternative way of representing the data - the total number of bytes sent or received by relays, divided by the number of *clients*, which is a constant parameter that is shared between both the paper and this thesis (namely, 2.5×10^6). We will also present the total relay bytes on a per-relay basis, as it will show us how much traffic each relay

handles in order to construct a circuit. A more client-centric metric is total client bytes. It presents the total number of bytes a client sends or receives per epoch.

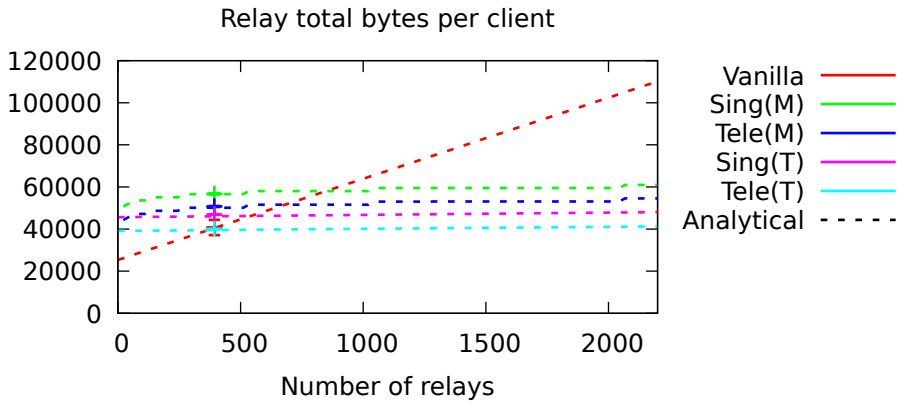
The next pages will present the results for each scenario. Recall that in section 4.1, we defined the following scenarios:

- *Walking Onions* - the results for this scenario have been obtained by running the simulator with a 5% sample of the full sample sets from 2019 and 2021 respectively;
- *Strolling Onions* - the results for this scenario have been obtained by running the simulator with a 5% sample of the filtered sample sets from 2019 and 2021 respectively. The filter applied was to exclude all bandwidths above the upper quartile from the sample set;
- *Running Onions* - the results for this scenario have been obtained by running the simulator with a 5% sample of the filtered sample sets from 2019 and 2021 respectively. The filter applied was to exclude all bandwidths below the upper quartile from the sample set.

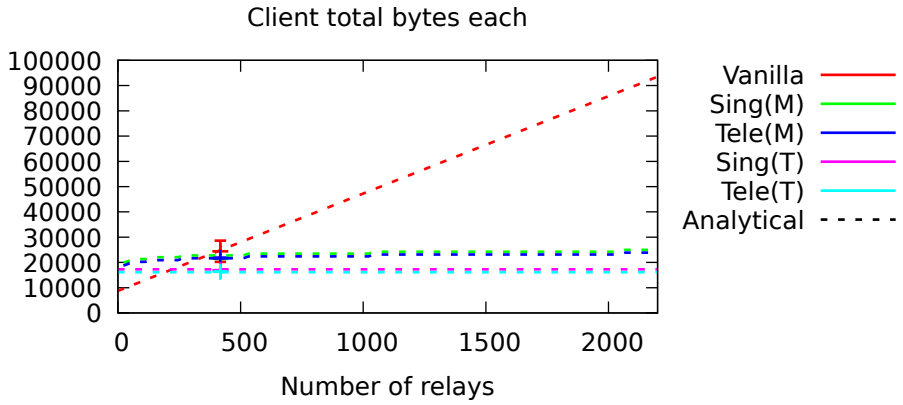
The labels in the graphs have the same meaning as in the original paper - (M) indicates Merkle authentication, whereas (T) indicates threshold signatures. The vertical black line represents the size of the current Tor network.

Scenario - Walking Onions

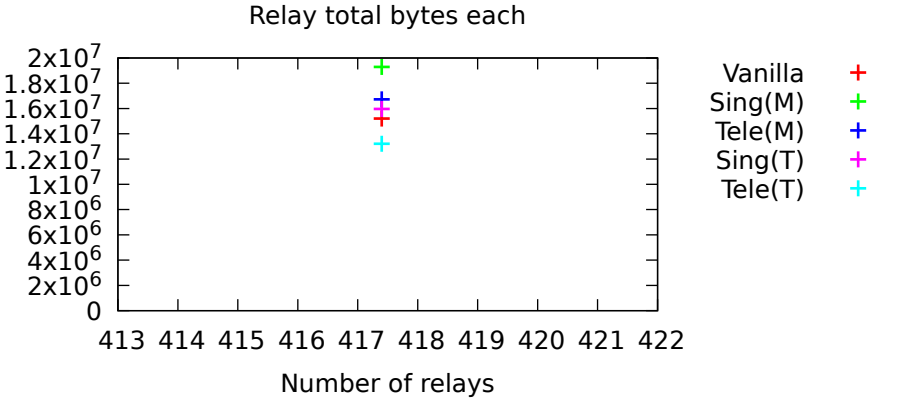
Figure 5.1: Walking Onions - 2019 sample set



(a) Total relay bytes per client

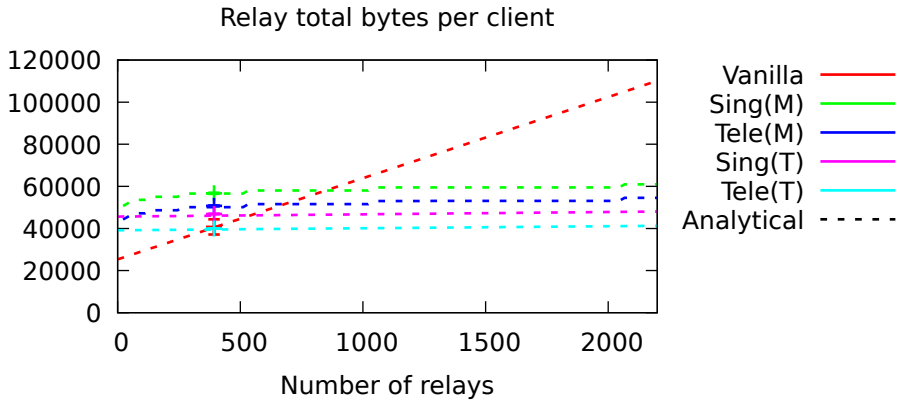


(b) Total client bytes

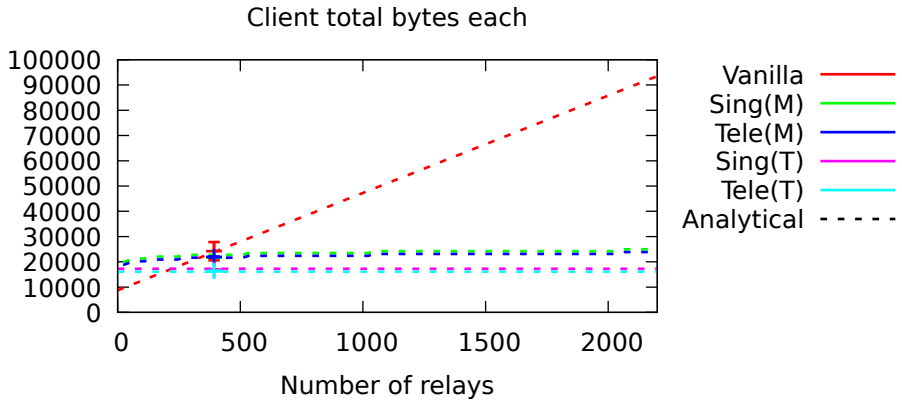


(c) Total relay bytes per relay

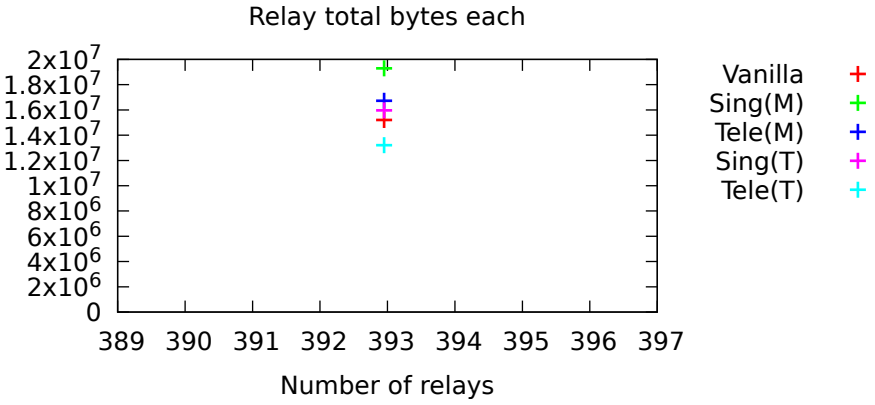
Figure 5.2: Walking Onions - 2021 sample set



(a) Total relay bytes per client



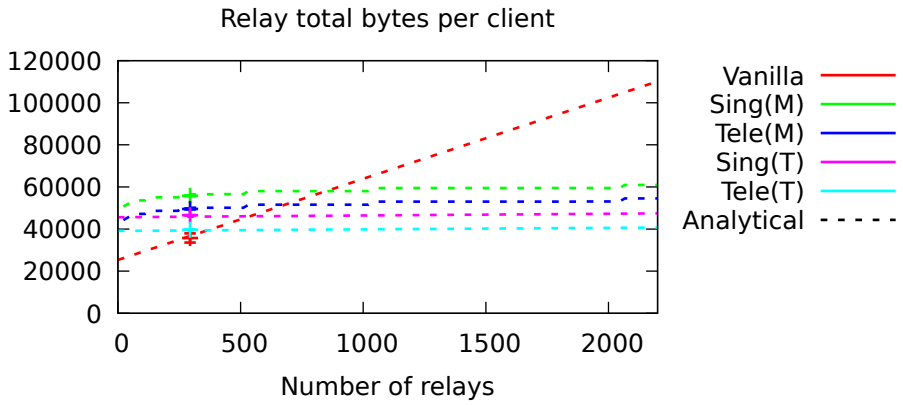
(b) Total client bytes



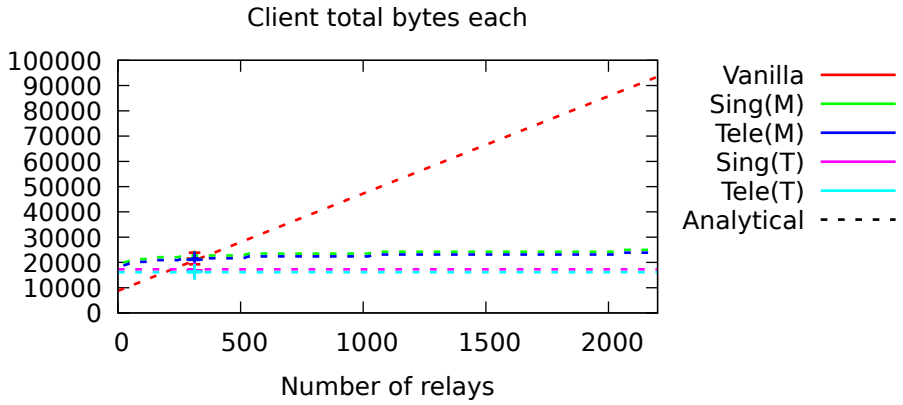
(c) Total relay bytes per relay

Scenario - Strolling Onions

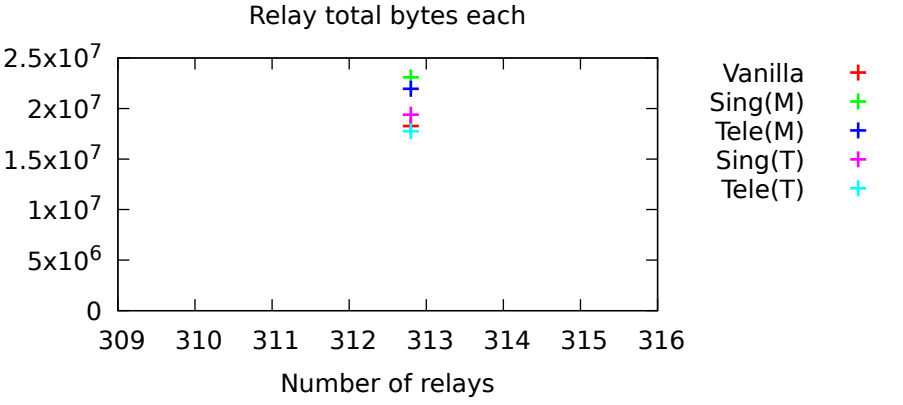
Figure 5.3: Strolling Onions - 2019 sample set



(a) Total relay bytes per client

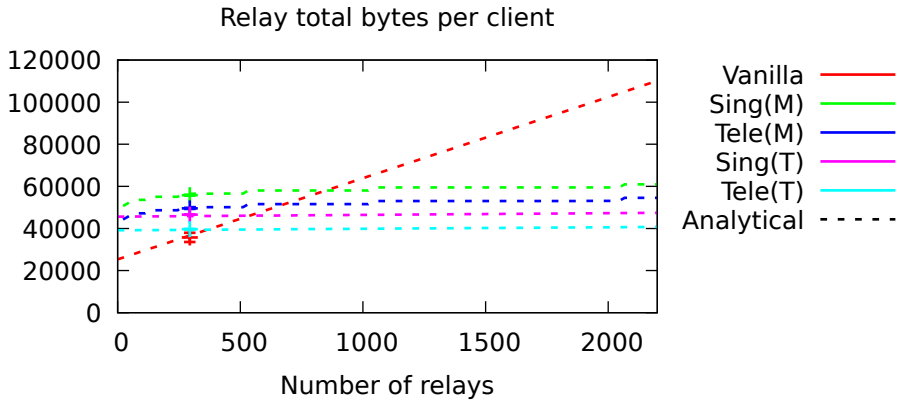


(b) Total client bytes

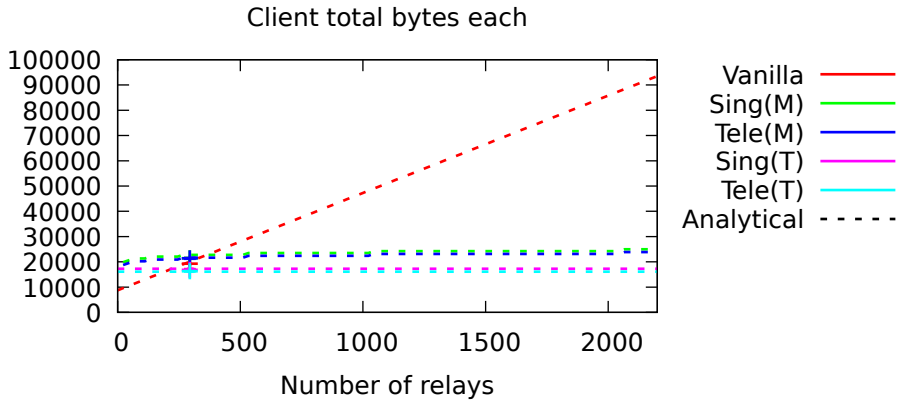


(c) Total relay bytes per relay

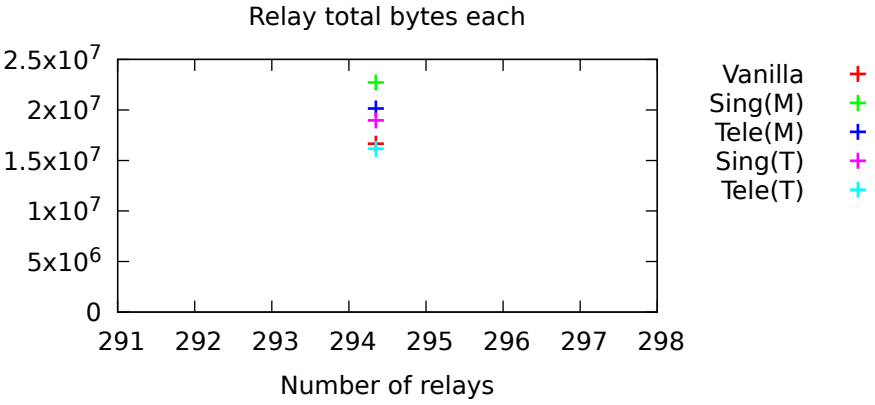
Figure 5.4: Strolling Onions - 2021 sample set



(a) Total relay bytes per client



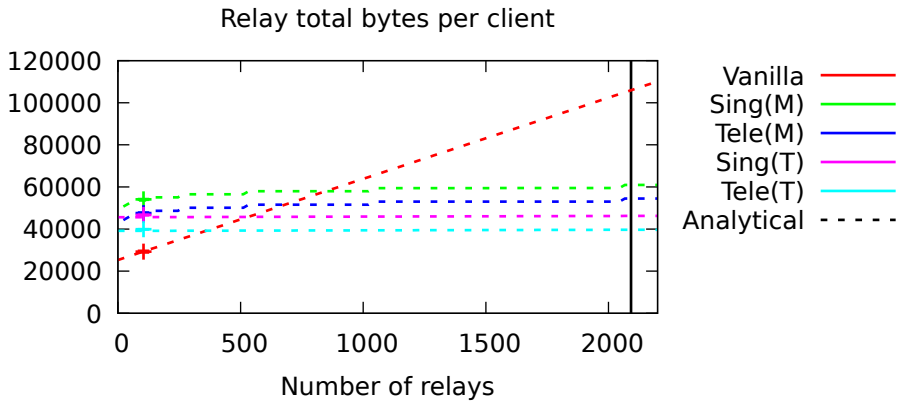
(b) Total client bytes



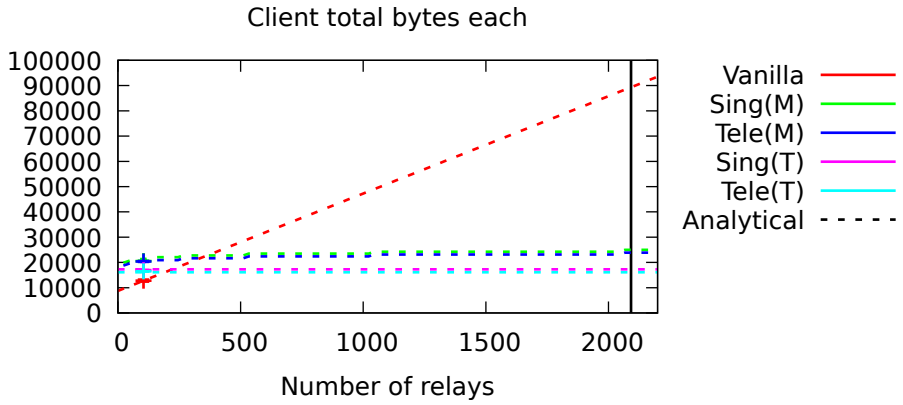
(c) Total relay bytes per relay

Scenario - Running Onions

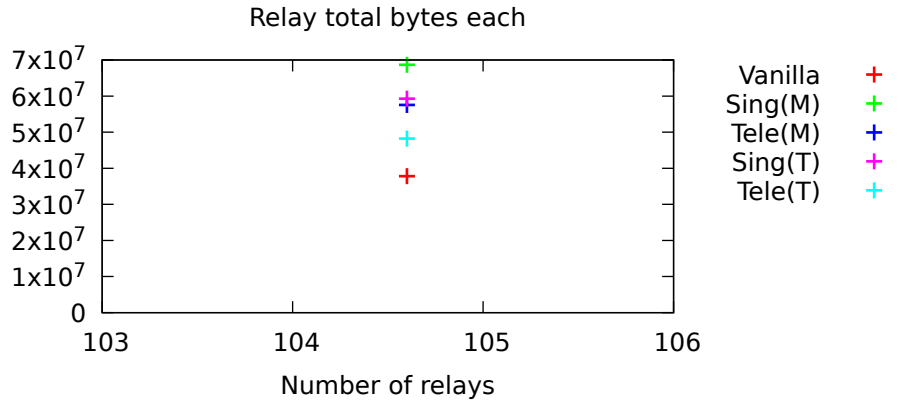
Figure 5.5: Running Onions - 2019 sample set



(a) Total relay bytes per client

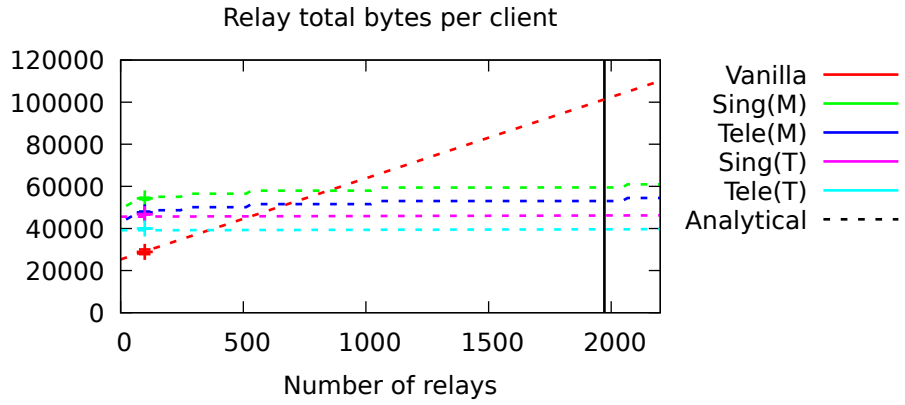


(b) Total client bytes

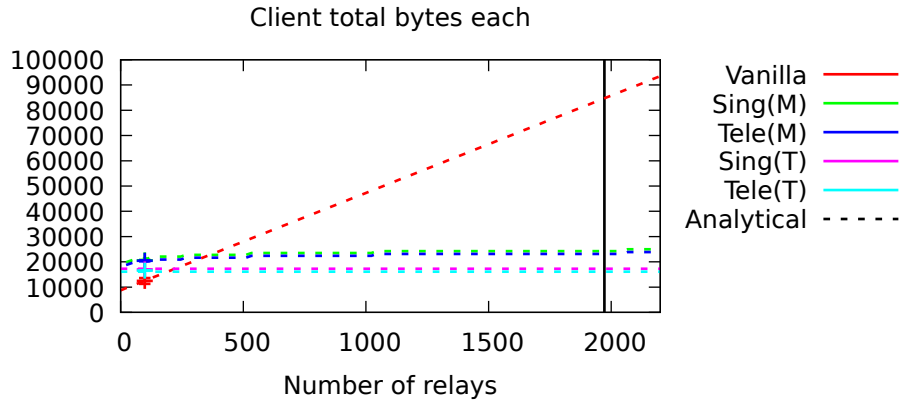


(c) Total relay bytes per relay

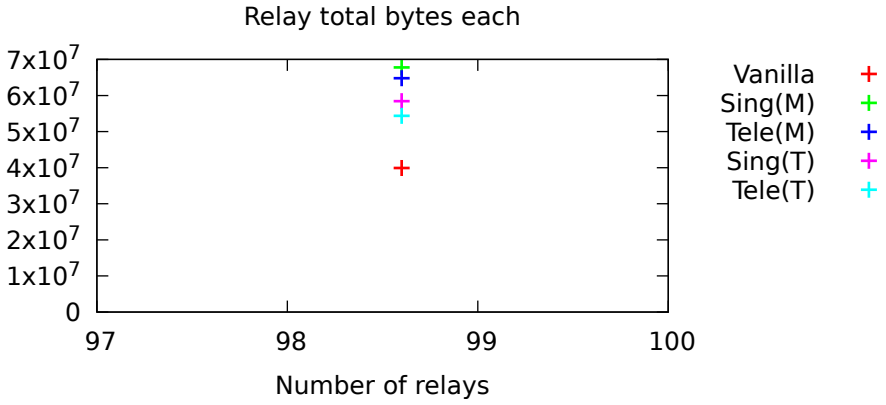
Figure 5.6: Running Onions - 2021 sample set



(a) Total relay bytes per client



(b) Total client bytes



(c) Total relay bytes per relay

The similarities between the results for relay total bytes per client and client total bytes in section 5.1 are clearly visible and this is to be expected. If there were any differences, they would point to the fact that the changes to the simulator have negatively impacted the calculation of the total bytes that pass through the relays and the clients or that the bandwidth of the relays affects the number of bytes needed for circuit construction. The graphs in figures 5.7 and 5.8 are from the original simulator and show the correctness of the applied changes - our simulation scenarios and the new bandwidth distribution algorithm do not affect the number of bytes sent or received.

Figure 5.7: Total relay bytes - original simulator

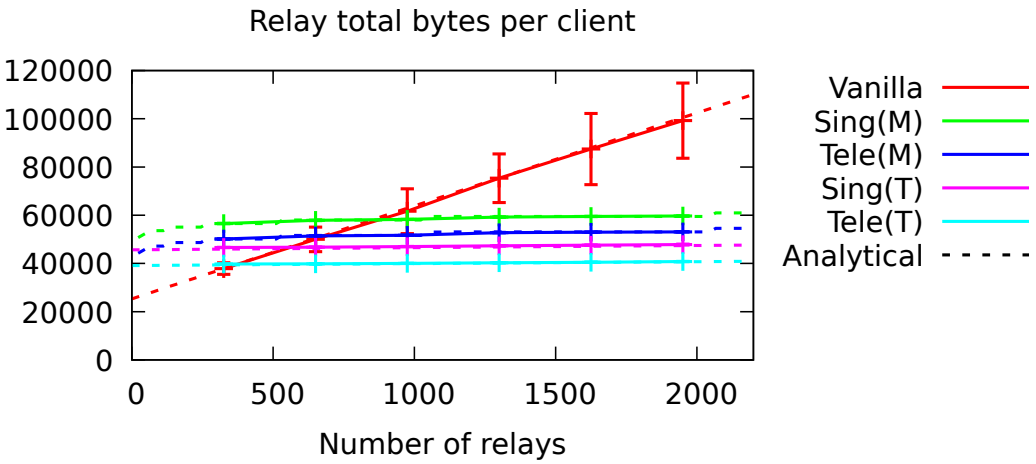
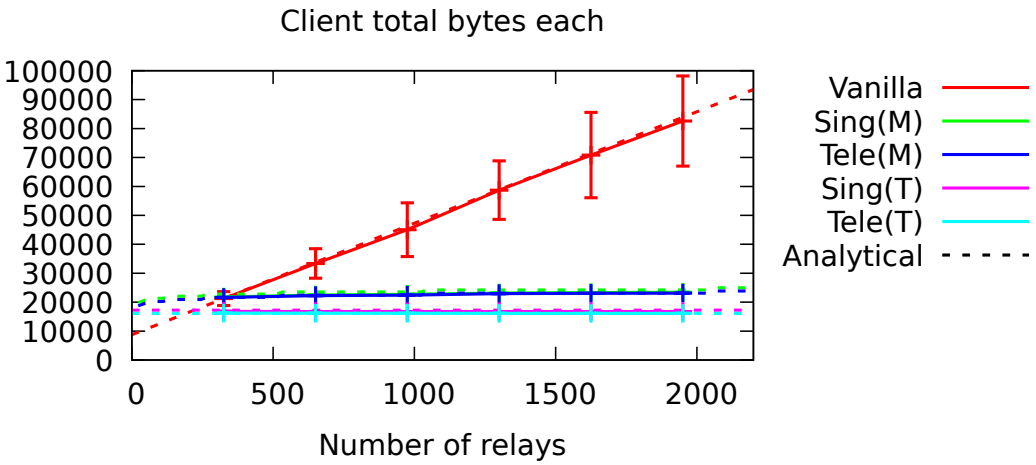


Figure 5.8: Total client bytes - original simulator



5.2 Total throughout

An important observation for the scenarios is the total throughput that the relays have per epoch. This enables us to judge how much of the total bandwidth has been consumed for circuit building alone. The total throughput (T) is calculated by the following formula: $T = S * 3600$, where S stands for the sum of the bandwidths of the current distribution in kilobytes per second and 3600 stands for the length of an epoch in seconds. Tables 5.1 and 5.2 represent the data obtained. The mean bandwidth and total throughput have been rounded down.

Table 5.1: Total relay throughput - 2019 sample set, 5% scale

Samples	**Number of relays**	**Mean bandwidth (kb/s, rounded)**	**Total throughput (kb, rounded)**
All samples	418	10669	1.606×10^{10}
Above $q_{0.75}$	105	34320	1.297×10^{10}
Below $q_{0.75}$	313	2617	2.950×10^{9}

Table 5.2: Total relay throughput - 2021 sample set, 5% scale

Samples	**Number of relays**	**Mean bandwidth (kb/s, rounded)**	**Total throughput (kb, rounded)**
All samples	394	17706	2.505×10^{10}
Above $q_{0.75}$	99	58393	2.081×10^{10}
Below $q_{0.75}$	295	4312	4.580×10^{9}

If we evaluate the results from section 5.1 together with the ones in tables 5.1 and 5.2, we can obtain the proportion of total traffic that was required for circuit construction for the different scenarios. As the values in the graphics are in bytes for all epochs, the relay traffic per epoch will be calculated as such: $S = N * T/(13 * (10^3))$, where T is the total bytes, N is the number of relays, 13 is the number of epochs (3 epochs are required for bootstrap) and 10^3 is required for conversion to kilobytes. For convenience, the evaluations are shown in tables 5.3 and 5.4. The evaluations have been obtained using the script `analysis/calcprop.py`, located in the simulator repository from the beginning of the chapter.

Table 5.3: Relay total bytes - 2019 sample set, 5% scale

Protocols	**Total throughput (kb, rounded)**	**Relay traffic (kb, rounded)**	**Proportion, rounded**
All samples			
Vanilla	1.606×10^{10}	8.879×10^{6}	5.530×10^{-4}
Tele(T)	1.606×10^{10}	7.985×10^{6}	4.974×10^{-4}
Tele(M)	1.606×10^{10}	1.002×10^{7}	6.242×10^{-4}
Sing(T)	1.606×10^{10}	8.602×10^{6}	5.358×10^{-4}
Sing(M)	1.606×10^{10}	1.030×10^{7}	6.416×10^{-4}
Below $q_{0.75}$			
Vanilla	2.950×10^{9}	8.792×10^{6}	2.980×10^{-3}
Tele(T)	2.950×10^{9}	8.543×10^{6}	2.896×10^{-3}
Tele(M)	2.950×10^{9}	1.056×10^{7}	3.581×10^{-3}
Sing(T)	2.950×10^{9}	9.332×10^{6}	3.164×10^{-3}
Sing(M)	2.950×10^{9}	1.111×10^{7}	3.767×10^{-3}
Above $q_{0.75}$			
Vanilla	1.297×10^{10}	6.083×10^{6}	4.689×10^{-4}
Tele(T)	1.297×10^{10}	7.764×10^{6}	5.985×10^{-4}
Tele(M)	1.297×10^{10}	9.257×10^{6}	7.136×10^{-4}
Sing(T)	1.297×10^{10}	9.537×10^{6}	7.352×10^{-4}
Sing(M)	1.297×10^{10}	1.105×10^{7}	8.520×10^{-4}

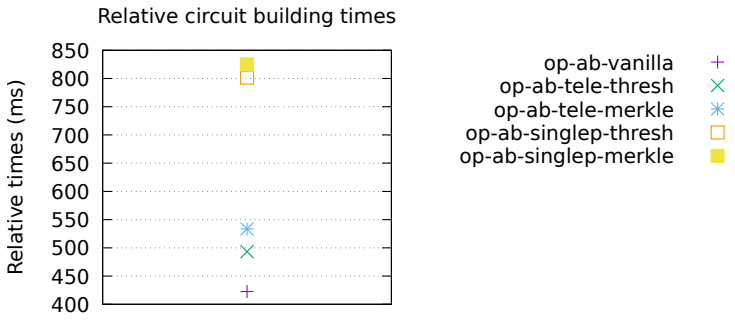
Table 5.4: Relay total bytes - 2021 sample set, 5% scale

Protocols	**Total throughput (kb, rounded)**	**Relay traffic (kb, rounded)**	**Proportion, rounded**
All samples			
Vanilla	2.505×10^{10}	9.195×10^{6}	3.670×10^{-4}
Tele(T)	2.505×10^{10}	7.989×10^{6}	3.189×10^{-4}
Tele(M)	2.505×10^{10}	1.011×10^{7}	4.037×10^{-4}
Sing(T)	2.505×10^{10}	9.651×10^{6}	3.852×10^{-4}
Sing(M)	2.505×10^{10}	1.166×10^{7}	4.656×10^{-4}
Below $q_{0.75}$			
Vanilla	4.580×10^{9}	7.541×10^{6}	1.646×10^{-3}
Tele(T)	4.580×10^{9}	7.317×10^{6}	1.598×10^{-3}
Tele(M)	4.580×10^{9}	9.123×10^{6}	1.922×10^{-3}
Sing(T)	4.580×10^{9}	8.589×10^{6}	1.875×10^{-3}
Sing(M)	4.580×10^{9}	1.029×10^{7}	2.246×10^{-3}
Above $q_{0.75}$			
Vanilla	2.081×10^{10}	6.055×10^{6}	2.909×10^{-4}
Tele(T)	2.081×10^{10}	8.247×10^{6}	3.963×10^{-4}
Tele(M)	2.081×10^{10}	9.826×10^{6}	4.721×10^{-4}
Sing(T)	2.081×10^{10}	8.863×10^{6}	4.259×10^{-4}
Sing(M)	2.081×10^{10}	1.028×10^{7}	4.941×10^{-4}

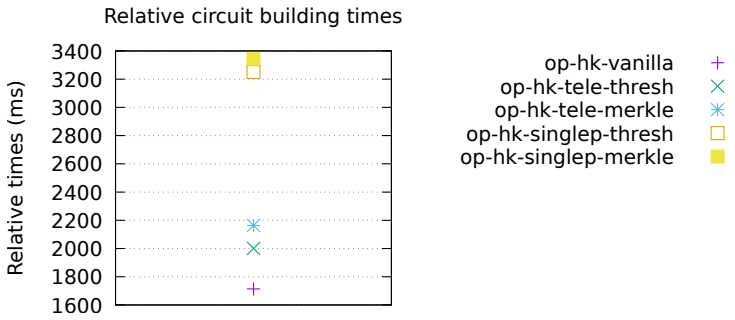
5.3 Circuit building times

Figures 5.9 and 5.10 present the relative circuit building times, calculated with algorithm 4.1. The points represent the mean times in milliseconds that are required for constructing a circuit, using the different protocols. As mentioned in chapter 4, the plots are based on data, obtained from the Tor metrics project. Under each plot, you will find the name of the OnionPerf [32] service that provided the data.

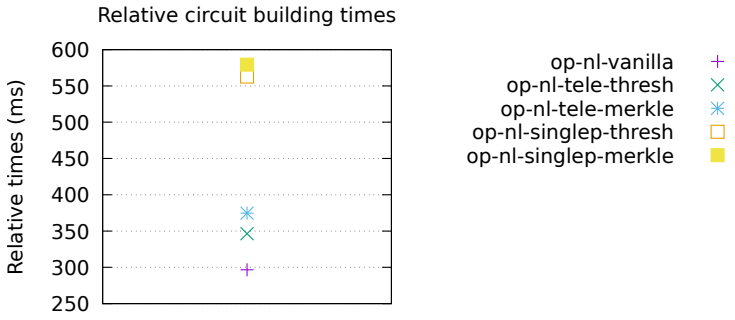
Figure 5.9: Relative circuit building times - 2019 sample set



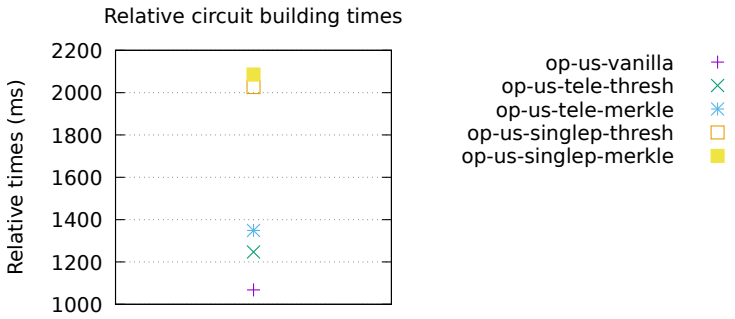
(a) op-ab



(b) op-hk

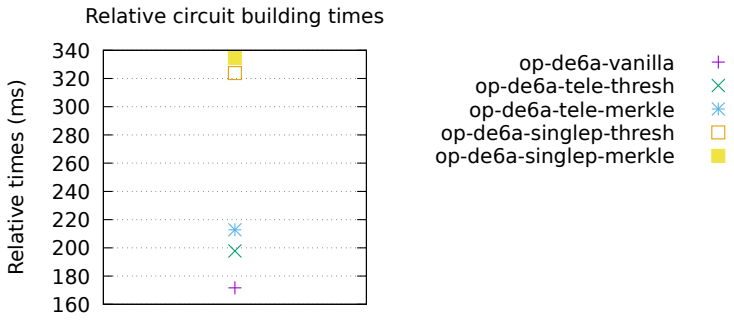


(c) op-nl

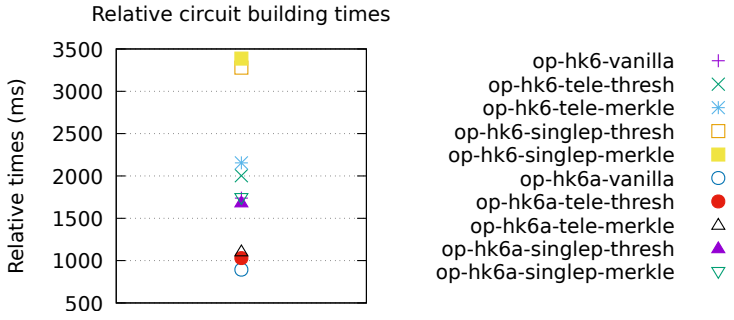


(d) op-us

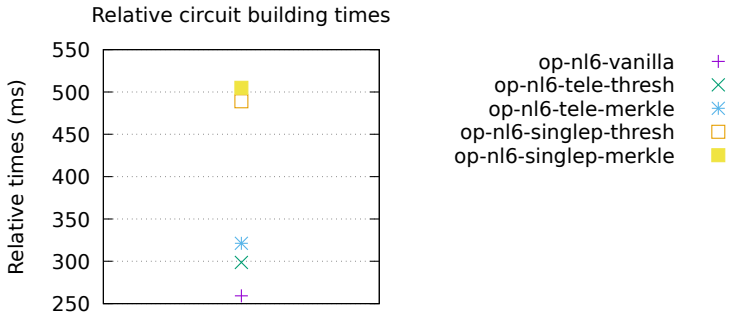
Figure 5.10: Relative circuit building times - 2021 sample set



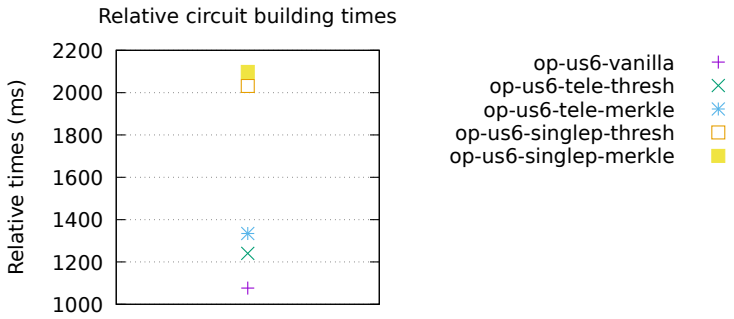
(a) op-de6a



(b) op-hk6



(c) op-nl6



(d) op-us6

CHAPTER 6

Discussion

The findings from chapter 5 indicate that because of the additional protocol complexity there will be trade-offs between the bandwidth used and the time needed for circuit construction.

Let us first focus on tables 5.1 and 5.2. The results show us that the proportion of traffic used for circuit construction at this scale (5% of the sample set) is negligible. This holds true for all three scenarios: all samples, samples under the upper quartile (global network slowdown), and samples over the upper quartile. An interesting observation is that the last scenario would actually reduce the proportion of construction-related traffic to total throughput when running the vanilla implementation as compared to the first one. Anonymity is, of course, compromised as per the reasons mentioned in section 4.1. Nevertheless, the analysis leads us to the conclusion that the *Walking Onions* set of protocols performs and scales objectively well on all devices, when the performance model used for evaluation is throughput.

While the original research focuses on the bandwidth used, which is a good usability mark, the thesis offers results that are based on a more user-centric metric - time. The data presented in section 5.3 indicates how much time a Tor implementation running the *Walking Onions* protocols would need for constructing a circuit in comparison with the vanilla implementation. It is visible that even with statistical interference considered, the improved stack builds circuits slower than the status-quo (thus confirming the hypothesis from 4.2.2). The current protocol needs the least time for circuit construction, single pass *Walking Onions* - the most, whereas the telescoping implementation is in the middle. Additionally, one can also see that clients, using Merkle trees as an authentication algorithm, generally need more time to obtain a functioning circuit than the ones using threshold signatures. The only exception is visible in 5.10 b), but it is due to the fact that two OnionPerf datasets were combined into one.

The facts from above suggest that there is a trade-off when using the improved protocol stack: while *Walking Onions* requires less bandwidth for constructing a connection, thus making space on the network for more actual data, the amount of computation needed impacts the circuit building times on slower devices.

When evaluating the results, it is important to keep in mind that the simulator and the study are not perfect. For example, these simulations were only run for 5% of the sample size due to hardware limitations. As such, possible future work would be to rerun the simulation on larger scale. Furthermore, while this experiment used an ideal-world simulator, written specifically for this use-case, an evaluation of the real-world protocol implementations also need been performed. Additionally, this work does not take into account statistical interference as per Jansen et al. [19], so performing sound evaluation is also a research possibility. Finally, as the drawbacks of the protocol can be mitigated to some extent by reducing the complexity, future work should focus on simplifying the algorithms without compromising security or anonymity.

CHAPTER 7

Conclusion

By conducting empirical tests in different simulation scenarios, based on real-world data, this study was able to prove that the *Walking Onions* set of protocols performs well in different environments. The results showed that the protocols would scale well with the Tor network and that the amount of bandwidth used for circuit construction remains small in comparison to the total throughput even if there is a global slowdown of the network. For all their positives, the protocols also have their negatives: the *Walking Onions* protocols will benefit the general throughput of the Tor network, but will damage the experience for users or relay operators, using under-powered devices like low-end smartphones or cheap single-board computers.

As each study, there have been some limitations - mainly hardware for simulations and statistical errors. Some improvements would be to rerun the simulation using larger sample sizes, run emulations with a real implementation of *Walking Onions* and count statistical interference in the results. Researchers and Tor developers should focus on improving the algorithms for circuit constructions as it would vastly improve the quality of life for users and relay operators.

List of Figures

List of Tables

List of Algorithms

Bibliography

[1] The Tor Project. Users - Tor Metrics. https://metrics.torproject.org/userstats-relay-country.html?start=2020-12-25&end=2021-12-25&country=all&events=off. Accessed: 23.02.2022.

[2] The Tor Project. How can I make Tor run faster? Is Tor Browser slower than other browsers? https://support.torproject.org/tbb/tbb-22/. Accessed: 23.02.2022.

[3] John Leyden. The 'one tiny slip' that put LulzSec chief Sabu in the FBI's pocket. https://www.theregister.com/2012/03/07/lulzsec_takedown_analysis/, 2012. Accessed: 23.02.2022.

[4] Chelsea H. Komlo, Nick Mathewson, and Ian Goldberg. Walking Onions: Scaling Anonymity Networks while Protecting Users. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1003–1020. USENIX Association, August 2020.

[5] Chelsea H. Komlo, Nick Mathewson, and Ian Goldberg. Simulator: Source code. https://git-crysp.uwaterloo.ca/iang/walkingonions. Accessed: 17.03.2022.

[6] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: The Second-Generation Onion Router. In *13th USENIX Security Symposium (USENIX Security 04)*, San Diego, CA, August 2004. USENIX Association.

[7] The Tor Project. Servers - Tor Metrics. https://metrics.torproject.org/networksize.html?start=2020-12-25&end=2021-12-25. Accessed: 23.02.2022.

[8] The Tor Project. Tor directory protocol, version 3. https://gitweb.torproject.org/torspec.git/tree/dir-spec.txt. Accessed: 23.02.2022.

[9] Dan Boneh, Ben Lynn, and Hovav Shacham. Short Signatures from the Weil Pairing. In Colin Boyd, editor, *Advances in Cryptology — ASIACRYPT 2001*, pages 514–532, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[10] Ralph C. Merkle. A Digital Signature Based on a Conventional Encryption Function. In *A Conference on the Theory and Applications of Cryptographic Techniques on Advances in Cryptology*, CRYPTO '87, pages 369–378, Berlin, Heidelberg, 1987. Springer-Verlag.

[11] Robin Snader and Nikita Borisov. A Tune-up for Tor: Improving Security and Performance in the Tor Network. In *Network and Distributed System Security Symposium*. Internet Society, February 2008.

[12] Steven J. Murdoch and Robert N. M. Watson. Metrics for Security and Performance in Low-Latency Anonymity Systems. In *Privacy Enhancing Technologies*, pages 115–132, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[13] Sebastian Müller, Franziska Brecht, Benjamin Fabian, Steffen Kunz, and Dominik Kunze. Distributed Performance Measurement and Usability Assessment of the Tor Anonymization Network. *Future Internet*, 4:488–513, 05 2012.

[14] SEOmoz Top 500. `https://moz.com/top500`. Accessed: 17.03.2022.

[15] Fatemeh Shirazi, Matthias Goehring, and Claudia Diaz. Tor Experimentation Tools. In *2015 IEEE Security and Privacy Workshops*, pages 206–213, 2015.

[16] Kevin Bauer, Micah Sherr, and Dirk Grunwald. ExperimenTor: A Testbed for Safe and Realistic Tor Experimentation. In *4th Workshop on Cyber Security Experimentation and Test (CSET 11)*, San Francisco, CA, August 2011. USENIX Association.

[17] Ken Yocum, Kevin Walsh, Amin Vahdat, Priya Mahadevan, Dejan Kostic, Jeff Chase, and David Becker. Scalability and Accuracy in a Large-Scale Network Emulator. *SIGCOMM Comput. Commun. Rev.*, 32(3):28, jul 2002.

[18] Rob Jansen and Nicholas Hopper. Shadow: Running Tor in a Box for Accurate and Efficient Experimentation. In *Proceedings of the 19th Symposium on Network and Distributed System Security (NDSS)*. Internet Society, February 2012.

[19] Rob Jansen, Justin Tracey, and Ian Goldberg. Once is Never Enough: Foundations for Sound Statistical Inference in Tor Network Experimentation. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 3415–3432. USENIX Association, August 2021.

[20] The Tor Project. Relay Associations. `https://community.torproject.org/relay/community-resources/relay-associations/`. Accessed: 23.02.2022.

[21] The Tor Project. Tor Metrics: circuit build times. `https://metrics.torproject.org/onionperf-buildtimes.html`. Accessed: 17.03.2022.

[22] The Tor Project. Tor Metrics: circuit build times spec. https://metrics.torproject.org/stats.html#onionperf-buildtimes. Accessed: 17.03.2022.

[23] Chelsea H. Komlo, Nick Mathewson, and Ian Goldberg. Simulator: Relay bandwidth distribution algorithm. https://git-crysp.uwaterloo.ca/iang/walkingonions/src/7d60eef9381137b4bec37921d7f7b862c543e4e4/simulator.py#L79. Accessed: 17.03.2022.

[24] Chelsea H. Komlo, Nick Mathewson, and Ian Goldberg. Simulator: Number of relays. https://git-crysp.uwaterloo.ca/iang/walkingonions/src/7d60eef9381137b4bec37921d7f7b862c543e4e4/simulator.py#L343. Accessed: 17.03.2022.

[25] The Tor Project. Tor Metrics: bandwidth files. https://metrics.torproject.org/collector.html#type-bandwidth-file. Accessed: 17.03.2022.

[26] The Tor Project. Simple Bandwidth Scanner: documentation. https://tpo.pages.torproject.net/network-health/sbws/. Accessed: 17.03.2022.

[27] The Tor Project. Tor Bandwidth File Format. https://gitweb.torproject.org/torspec.git/tree/bandwidth-file-spec.txt. Accessed: 23.02.2022.

[28] Rob Jansen, Kevin Bauer, Nicholas Hopper, and Roger Dingledine. Methodically Modeling the Tor Network. In *5th Workshop on Cyber Security Experimentation and Test (CSET 12)*, Bellevue, WA, August 2012. USENIX Association.

[29] Python Software Foundation. Python: multiprocessing library. https://docs.python.org/3/library/multiprocessing.html. Accessed: 17.03.2022.

[30] Python Software Foundation. Python: Process class. https://docs.python.org/3/library/multiprocessing.html#multiprocessing.Process. Accessed: 17.03.2022.

[31] Python Software Foundation. Python: resource library. https://docs.python.org/3/library/resource.html. Accessed: 17.03.2022.

[32] The Tor Project. Tor GitLab: OnionPerf. https://gitlab.torproject.org/tpo/network-health/metrics/onionperf. Accessed: 17.03.2022.