FACULTY OF SCIENCE, TECHNOLOGY AND COMMUNICATION

Hidden Service Tracking Detection and Bandwidth Cheating in Tor Anonymity Network

Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of Master in Information and Computer Sciences

Author:
Fabrice THILL

Supervisor:
Prof. Alex BIRYUKOV

Reviewer:
Prof. Volker MÜLLER

Advisor:
Ivan PUSTOGAROV

August 2014

Acknowledgements

I would like to thank: Prof. Biryukov for giving me the opportunity to work on this topic; Ivan Pustogarov for his help and guidance; the University of Luxembourg for providing servers used in the experiments.

Abstract

Instantaneous online communications through the Internet have become an integral part of our world, both for business as well as for individuals. However, some entities restrict access to parts of the Internet, in an effort to establish censorship, while others monitor online communications. To circumvent censorship, restore privacy and anonymity, some tools such as Tor, a low latency anonymity network, have been created.

This work focuses on Tor and has two contributions:

1. A method of detecting tracking attacks on hidden services as well as an implementation. By analyzing 39841 hidden service addresses for the year 2013 we found that at least 50 Tor relays were conducting tracking attacks on a total of 45 hidden services.

2. An attack on the Tor bandwidth measurements protocol in order to obtain higher chances of being chosen for a client path, the implementation reaching a 1% probability to be chosen as an Exit node on the Tor network and the 21^{st} highest relay, in terms of bandwidth weight, out of 5.834 active relays, while using a bandwidth only 50 KB/s.

Contents

Chapter 1

Introduction and Background

1.1 Foreword

Tor is a low latency anonymity system originally developed by the U.S. Naval Research Laboratory [14], now an open source project developed by "The Tor Project", a non-profit organization [13]. It is nowadays the most popular anonymity and censorship circumvention tool as well as the most researched onion routing scheme. Originally the Tor network was comprised of about 40 nodes with a bandwidth usage of about half a gigabyte a week. At the time of this writing the Tor network is comprised of more then 5000 nodes with the top nodes relaying multiple terabytes daily. Some of the success of Tor can be attributed to the goals defined in the original design paper [8]. These goals were: deployability, usability, flexibility and a simple design. The Tor browser bundle, a version of the Firefox web browser modified to provide better security and automatically route traffic through the Tor network client, is a manifestation of these goals. The Tor threat model does not include a global adversary, i.e. an adversary which can listen, block or modify all messages between routers. Instead Tor is built to defend against an attacker, which can only listen to part of the traffic and can possibly corrupt or control some Tor nodes.

The next chapter will first give a broad explanation of Tor, then focus on the Hidden Services, a functionality which provides responder anonymity and finally explain the consensus bandwidth weights, motivations, how the weight is obtained and how it affects path creation in Tor.

The second chapter will provide methods and an implementation to detect tracking attacks of hidden services, along with the experimental results of our study. Tracking service activity and usage as well as censoring is achieved by gaining control over responsible directory nodes. The possibility of such attacks was predicted in [9]. The actual attacks were demonstrated in "Trawling for Tor Hidden Services: Detection, Measurement, Deanonymization " [7].

The third chapter will focus on how the Tor bandwidth measurement protocol can be abused in order to obtain a higher chance to be chosen by a client, thus facilitating other attacks such as traffic confirmation [22] and website fingerprinting [23].

Finally, the last chapter will discuss some of the obtained results and propose countermeasures.

1.2 Tor: Generation 2 Onion Routing

Tor is an onion routing system, this means that instead of a directly connecting to a website or service, client sends his traffic through multiple servers in order to conceal his IP address. Originally Tor was an acronym for *The Onion Router*, the term *onion routing* comes from the layers of encryption around the connection.

Assume Alice wants to send a message to a website using the Tor network. Alice uses a Tor client or Onion Proxy (OP) which will relay the message through the Tor network. When the OP is first connected to the network, it downloads directory information from the directory authorities. The authorities are a small set of trusted nodes (currently 9) in the network which gather information on each relay in the network.
Once the OP has learned enough information about relays, it can start building a *circuit*, i.e. a path through the Tor network. All traffic in Tor is exchanged using cells of fixed 512 byte size. Each cell contains a circuit-id, a command and a payload. The circuit-id is a randomly chosen number which links together two connections on a relay. Commands can be classified into two categories, control and relay commands. Control commands are meant for the relay that receives them to be interpreted; relay commands instruct a relay to forward a cell to the next node in the circuit. In order to create a circuit, the OP first establishes a TLS connection with the first node (also known as *Entry node* or *Guard node*). It then sends a `CREATE` cell to the Guard node over TLS. The cell is additionally encrypted using the public key of the first node and contains the first part of the Diffie-Hellman key exchange handshake. The Guard node responds with the second half of the Diffie-Hellman handshake.

Now that a symmetric key has been created between the two parties, the OP sends an `EXTEND` cell to the Guard node, again containing the first half of the Diffie-Hellman handshake encrypted with the public key of the second node (also known as *Middle node*). The Guard node extracts the payload, puts it into a `CREATE` cell and sends the cell to the Middle node. The Middle node responds with the second half of the Diffie-Hellman handshake, which is relayed to the OP by the first node.
This step is repeated a third time for the Exit node, the last node in the 3 hop circuit Tor uses by default for clients. Alice's OP now shares a symmetric key with each of the relays and can now use the circuit to send and receive information.

In order to send a message to a website for instance, the OP creates an encrypted message with the structure : $C_1, \{\{\{, m\}_{k_3}\}_{k_2}\}_{k_1}$, where $\{X\}_{k_i}$ denotes AES encryption with key k_i, and C_1 denotes circuit ID. It then sends the encrypted message to the first relay, with which is shares the symmetric key k_1. The first relay removes a layer of encryption and obtains the circuit ID C_1 and the encrypted message $\{\{m\}_{k_3}\}_{k_2}$ which it cannot decrypt. With C_1 the relay knows that it should send the message to the second relay and does so, while providing C_2 for the next relay. This step is repeated, each time removing a layer of encryption, until the message reached the Exit node, which can remove the final layer of encryption, learn the final destination and send the message to the website.
Figure 1.1 was taken as is from [8] and shows one such circuit creation as well as the subsequent data transfer between the client and the website.

If there is a response from the website, the functioning is the same, but instead of decrypting, each relay adds a layer of encryption. When this encrypted message arrives at the sender, it can decrypt all 3 layers and read the response. In essence, this system provides a method of breaking the link between the message and the sender. The Guard node has sender's IP

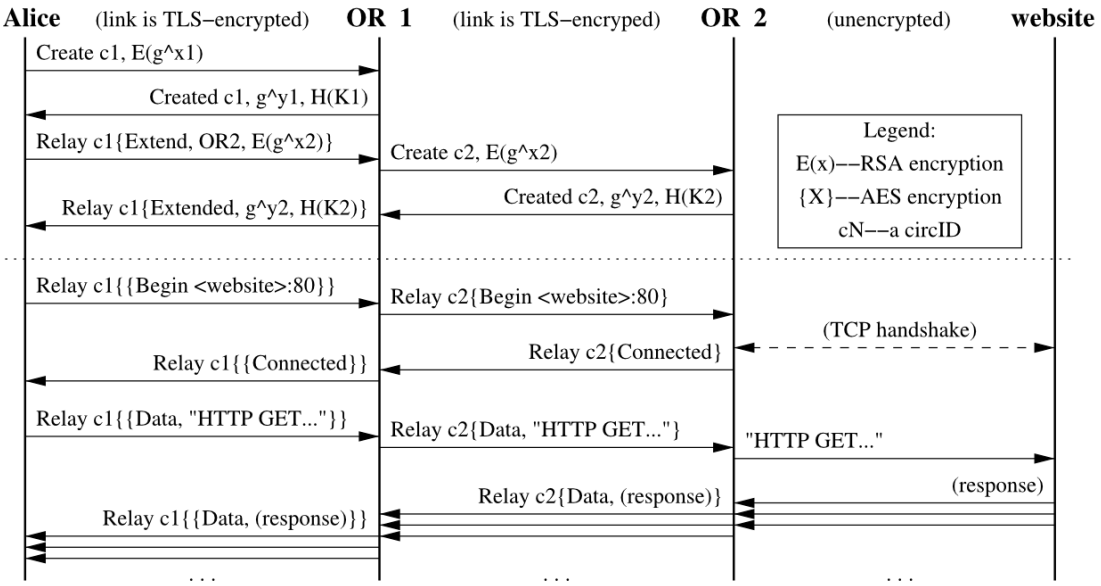

Figure 1.1: A two-hop circuit creation with data exchange

address but no information on the message or its destination, while the Exit node has both message and the destination, but does not learn any information about the sender.

It can also be noted that should an attacker be able to read both data entering the Guard node and the data exiting the Exit node, or possibly control both nodes, it is possible to do traffic correlation attacks which deanonymise the user[1].

Tor in its current implementation by default uses 3 hop circuits, meaning a relay can be either a Guard node, Middle node or Exit node. Each type of node has different requirements:

Guard nodes. Guard nodes are arguably the nodes in which the most trust is placed since they protect user's anonymity. Possible attacks against the Tor network include correlation attacks (in which the attacker compares traffic coming to the Tor network with traffic leaving the network) and website fingerprinting attacks (where the traffic of users is compared to pre-build patterns) .

If an attacker controls A out of N nodes in the network[2], the probability of the attacker controlling both the entry node and the Exit node of a random circuit in the network is $\frac{A}{N}^2$, but if the client over times builds a lot of different circuits, the probability of the attacker controlling one pair of entry router and exit router over the possible pairs converges rapidly to 1.
To prevent this from happening each client chooses Entry nodes from a set of three Guard nodes. A Guard node remains in the set for a random duration between 30 and 60 days[3]. The fact that a Guard node is chosen at the beginning of the rotation period only (and not for each circuit) and remains the same during the whole period dramatically reduces the convergence

[1]By deanonymisation we mean revealing the user's public IP address. In case of a user behind NAT (the most common case), this is an IP address of the user's ISP.

[2]The probability for a relay to be chosen by a client is proportional to its bandwidth, however for the sake of simplicity of the example we assume that the relay's bandwidth is not taken into account.

[3]However at the time of writing Tor developers are considering switching to only one Guard node with a rotation period of several months.

rate. The probability of choosing an attacker node at the beginning of a rotation period is $\frac{A}{N}$. This results in that the attacker controls either 1/3 of the user's circuit for the duration of the period or none of them.
A node is only used by clients as a Guard node if it has a Guard, Valid and Running flag.
An explanation on all the flags can be found in section 1.3

Middle nodes. There are no requirements for being a Middle node except of course having the Valid and Running flag. Middle nodes are the nodes in the network which learn the least amount of information about clients since they only relay encrypted information from one relay to the next.

Exit nodes. Exit nodes are relays which allow connections outside the Tor network on behalf of a client. A Tor Exit node operator can restrict the addresses and ports to which it allows outgoing connection in the relay's Exit policy. Exit nodes require an Exit, Valid and Running flag (see section 1.3). Controlling both Guard and Exit nodes is crucial for a traffic confirmation attack.

1.3 Network Consensus Documents

The Network Consensus Documents (hereafter simply called consensus document or consensus), are files containing the results of all votes by the directory authorities on the status of all active relays in the network. A consensus is produced every hour and is valid for a period of 3 hours.

We now give an overview over the consensus information necessary for this work; additional information can be found in the Tor specification document [20]. The latest consensus as well as the archives for all public data on Tor can be downloaded from [17].

A consensus begins with general information such as version, validity time, known flags as well as parameters for votes and information on each directory authority.
For each relay in the consensus we have multiple lines each starting with a letter :

- r : General information
  - The name, which serves as an identity readable by humans; it does not however have to be unique
  - The fingerprint or identity, a base-64 encoded hash of the public key
  - The digest, a base-64 encoded hash of the most recent descriptor
  - The publication time of the most recent descriptor
  - The IP address (IPV4)
  - The OR port, the port the onion port uses for its communications
  - The directory port, the port used to transmit directory information; the directory port is optional and is 0 if not present
- s : Flags, if present then:

  - Valid, the relay has a valid Tor version, and is not blacklisted by the authorities
  - Named, the name can be mapped to a unique fingerprint; it must have an uptime of at least 2 weeks
  - Unnamed, the name cannot be mapped to a unique fingerprint
  - Running, an authority was able to establish a connection to the relay within the last 45 minutes
  - Stable, the router's mean time between failures is either: (1) at least median for known active routers or (2) more then 7 days
  - Exit, the exit policy allows to exit to at least two out of the three ports 80, 443 or 6667, and allows a connection to at least 1 /8 address subspace.
  - Guard, the router needs to have an uptime that is at least median of all familiar nodes and needs to have a bandwidth of at least median or 250 KB/s. A node is familiar, if at least 1/8 of the nodes are more recent, or if it has been up for a few weeks.
  - Fast, the router is in the top 7/8 or a bandwidth of at least 100 KB/s
  - Authority, the router is a directory authority
  - V2Dir, the router supports the version 2 directory protocol and has a directory port
  - HSDir, the router has been up for at least 25 hours and stores and serves hidden service descriptors

- v : Tor version
  - The Tor version that the relay is currently running
- w : Bandwidth information
  - The bandwidth consensus weight, see section 1.5. If not yet measured, it is set to 20 and followed by an additional "unmeasured=1"
- p : Exit policy
  - The exit policy of the relay. A relay administrator may choose to block or allow specific ports and addresses. If the exit policy is reject *:* the relay will not serve as an Exit node.

1.4 Hidden Services

Tor was originally designed to provide client anonymity, but later added a feature to provide responder anonymity. This feature was named "Hidden Services" and permits running services, for instance a web service, without the client ever learning its IP address. As a result the physical location of the web server is protected and it becomes harder for an entity to block or hinder the service from functioning properly. This property is achieved by having both parties interact through a relay in the Tor network called a Rendezvous Point (RP).

Additionally to providing responder anonymity, Tor also provides two ways of setting up a hidden service with client authorization, meaning only a selected group of clients will have access to the hidden service.

The first method utilizes a secret key and cannot prevent clients from learning further hidden service activity once they know the key.

The second method can prevent unwanted users from even learning the introduction points, but requires that the hidden service sends a different encrypted list of introduction points to each single client separately and is thus only feasible for very small groups of clients, up to 16 according to the documentation [19].

1.4.1 Rendezvous protocol

We now give an outline of the most recent protocol version without client authorization; further information as well as older and partially still used protocol versions can be found under [19]. For this document, unless specified otherwise, all hashes are SHA-1.

With Alice as the client and Bob as the respondent, the protocol outline is as follows :
Bob links the service he wants to offer with his Onion Proxy (OP). The Onion Proxy is a Tor instance, which main purpose is to serve as a proxy, meaning it will act as a trusted middle man between the service and the Tor network. All data between the Tor network and the service will pass through the OP. When the OP is first started, it automatically generates a private/public key pair, which is done by every Tor instance. Next a set of Introduction Points (IP) are chosen. IP's are Tor relays which, additionally to their normal function, will later store information about the hidden service and make a meeting at the Rendezvous Point (RP) possible. In the current Tor implementation Bob's OP chooses 3 distinct IP's. Bob may choose not to advertise his hidden service to an IP in order to make his service unreachable.

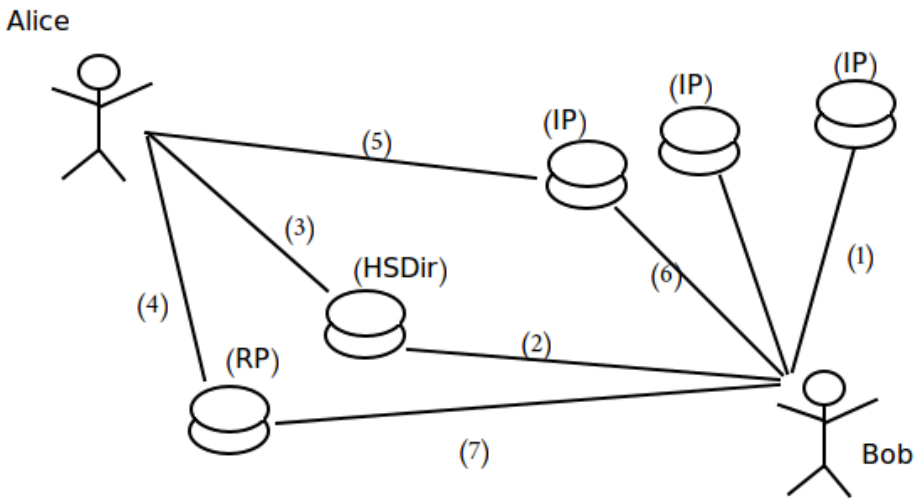

Figure 1.2: Tor rendezvous protocol

Bob's OP then creates a circuit to each IP (1) and sends a `RELAY_COMMAND_ESTABLISH_INTRO` cell with a payload containing the length of the public or service key, the public/service key itself, a hash of the sessions information in order to prevent replay attacks and finally a signature of the information just listed. The service key is a public key associated to the hidden service as opposed to the OP.

When an IP receives this cell, it first verifies if the hash is correct, which it can do since the session information is shared between both parties. It then verifies the signature, which it can do since the key is contained in the message itself. If both checks are successful, it returns an empty `RELAY_COMMAND_INTRO_ESTABLISHED` cell and drops all other circuits associated with Bob's public key, so that there is only one circuit between the Onion Router (OR) and Bob's OP.

Then the OP generates the service descriptors. V2 service descriptors contain the following information [4]:

- "rendezvous-service-descriptor" followed by a descriptor-id, where the descriptor-id is calculated as in algorithm 1
- "version", the version of the protocol being used
- "permanent-key", the key needed to verify the descriptor-id and the signature
- "secret-id-part", the secret-id as calculated in algorithm 1 for verification purposes
- "publication-time", the time in format YYYY-MM-DD HH:MM:SS, a timestamp of the descriptor creation rounded down to the nearest hour
- "protocol-versions", the string containing all versions which can be used
- "introduction-points",
  a possibly encrypted string containing information on the introduction point, such as: identifier, IP address, port, onion key, service key
- "signature" followed by Bob's signature of the message on a new line

---

Data: Input : **Permanent id** - a hash of the services public key truncated after 80 bits
replica - an integer, in the current implementation either 0 or 1 (thus two descriptor-ID's will be generated)
time - a Unix time stamp
descriptor cookie - optional, 128 bits secret password
Result: descriptor-id
initialization: validitytime = 86400 (24 hours);
timeperiod = ((time+first byte of permanent id * validitytime)/256)/validitytime;
secret-id = H(time-period | descriptor-cookie | replica);
descriptor-id = H(permanent-id | secret-id);
return base32 encoding of descriptor-id;

Algorithm 1: The Computation of Descriptor-ID for a Hidden Service

---

Once the service descriptors have been created, (2) Bob's OP advertises the service by uploading the hidden service descriptors to a subset of Hidden Service Directories (HSDir's). The HSDir's are placed in a circular list ordered by their fingerprints, and for each descriptor-id, the 3 relays following become responsible HSDir's for a period of approximately 24 hours.

[4] Each item is a new line, all characters between "" are literal strings found in the descriptor followed by a value, here followed by a description of the value.

The time period is only approximately 24 hours, because of the addition of the first byte from the permanent id in the calculation of the descriptor-id in algorithm 1. This is done to prevent having all hidden services changing their descriptor-id at the same time.

Since in the current Tor implementation two descriptor-ids are generated, subset of six HSDir's are responsible for each time period. By looking at the consensus, Bob's OP can extract all HSDir's, since hidden service directories are marked by the *HSDir* flag. After extracting all HSDir's, the OP can sort the HSDir's by fingerprints to determine to which HSDir it should send the descriptors.

Each hour, the OP generates two new descriptors and uploads them to the correct HSDir's. When a HSDir receives a descriptor, it checks :

- if the signature is invalid
- if the descriptor is older than 24 hours
- if the descriptor starts its valididty more than 1 hour in the future
- if the HSDir already has a more recent descriptor

If anything applies, the descriptor is rejected, else the HSDir remembers it for 24 hours.

Through one of the hidden service wikis[4] or other traditional means (blogpost, email etc.), (3) Alice learns about a hidden service in a z.onion address format, where z is the base-32 permanent identifier for the hidden service. Alice's OP can then fetch the hidden service descriptors by opening a circuit to an appropriate HSDir. The descriptor is retrieved by making an HTTP 'GET' request to "/tor/rendezvous2/<z>". The OP tries any of the 6 HSDir randomly, until encountering a usable HSDir or until all 6 get request have failed.

If Alice's OP successfully retrieves a descriptor-id, and if she is able to guarantee its validity (by checking the signature, time stamp etc..), she can then open a connection to a Rendezvous Point (RP) (4), randomly chosen by her. The OP does so by creating a circuit to the RP and sending a `RELAY_COMMAND_ESTABLISH_RENDEZVOUS` cell which contains a *Rendezvous Cookie*. In case of success, the RP responds with `RELAY_COMMAND_RENDEZVOUS_ESTABLISHED`. Alice's OP should use a new cookie for each connection attempt and only use the circuit for communication with the rendezvous point.

Next Alice's OP sends a `RELAY_COMMAND_INTRODUCE1` cell to 1 out of the 3 IP's (5) with the payload containing Bob's service key, information on the RP as well as the Rendezvous cookie and the first part of Diffie-Hellman key exchange handshake (g^x). If the IP recognizes the service key, it sends `RELAY_COMMAND_INTRODUCE2` on the appropriate circuit to Bob (6) and replies to Alice with `RELAY_COMMAND_INTRODUCE_ACK` or discards the cell and replies with an error message, if the service key is not valid.
Bob's OP decrypts the payload and, should he choose to, creates a circuit to the RP and sends a `RELAY_COMMAND_RENDEZVOUS1` with the payload containing the rendezvous cookie, the Diffie-Hellman response (g^y) and a handshake digest. If the RP recognizes the rendezvous cookie, it relays the rest of the body to Alice with a `RELAY_COMMAND_RENDEZVOUS2` cell.
When receiving the cell Alice can finish the handshake and open a TCP stream by sending a `RELAY_COMMAND_BEGIN` cell.
If successful (7), they may now communicate with the RP relaying messages between the two circuits.

1.4.2 Hidden Service Usage Tracking and Censoring

The fact that descriptor-ids are being used to determine the responsible HSDir and that descriptor-ids can be calculated in advance, means that by finding the appropriate public/private key-pair, a hidden service directory can position itself on the distributed hash table, in such a way that it will have a high probability of becoming responsible HSDir on a wanted date. Note that this is not computationally expensive.

It is this property that is used in [7] to collect information on hidden services since by controlling one responsible HSDir it is possible to count the number of incoming connections and to extrapolate the number of users of a particular hidden service.
If an entity is responsible for all 6 HSDir's, it can censor the service by returning an error cell each time a client tries to fetch descriptor information.

1.5 Consensus Bandwidth Weight

Tor is a volunteer-based network and anybody can set up a Tor relay. Running Tor relays announce their existence by posting their descriptors to a small set (currently 9) of trusted Tor authorities. The authorities periodically vote and assemble a list of reachable Tor relays called *Network Consensus Document* or shorthand *consensus*. Each hour, a new consensus is created, by taking the majority vote of all information the authorities believe represent the state of the network. Tor clients then fetch consensus from the authorities.
Each relay in the consensus is assigned a bandwidth weight which is used by clients in the path selection algorithm. Roughly, the probability for a relay to be chosen as a part of a circuit is proportional to its consensus bandwidth weight. In this way bandwidth weight become crucial when mounting a traffic correlation attack as well as for load balancing.
Initially Tor authorities used relays self-advertised bandwidths capped by 10 MiB/s. This however resulted in poor load balancing and allowed an attacker to easily cheat about their bandwidth. Later *bandwidth scanners* were introduced, which do active measurements to determine a relay's unused bandwidth.

This section gives an overview of Tor bandwidth scanners. It explains how the bandwidth measurements work, how the results are aggregated to produce the bandwidth consensus weight and finally how clients choose their path for circuit creation in Tor and how it is influenced by the weight.

Information in this sections was extracted from the Tor bandwidth specification document [16] and the corresponding source code [21](which are not completely in concordance). The complete information on the path finding logic for Tor can be found in the Tor path specification document [18].

1.5.1 Bandwidth Scanners

In their essence bandwidth scanners are python scripts run by Tor authorities, which continuously download large files through 2-hop circuits in order to assess how much unused bandwidth Tor relays have. As of July 2014, by looking at the consensus votes documents, 4 out of the 9 authorities seem to be running bandwidth scanners. These 4 are: gablemoo, moria1, tor26 and maatuska.

In the default configuration any authority that wants to run bandwidth scanners will have 4 scanners running in parallel, each one measuring a different partition of the Tor network. According to the specification documents the first scanner measures the 12 fastest percent, the second one measures the nodes in the 12 to 35 percent slice, scanner 3 goes from 35 to 60 percent and the last one takes the 40 slowest percent. The torflow source code [21], which is the name of the project containing the bandwidth scanners, shows a segmentation of 10%, 10-40%, 30-60%, 60-100%.

All 4 scanners use the same Tor client, to which they connect using the Tor control port, a protocol, implemented as a telnet service, used in order to communicate with a running Tor instance. A user can subscribe to different events by sending a `SETEVENTS` command to the control port. Bandwidth scanners subscribe to the following events: `NEWCONSENSUS`, `NEWDESC`, `CIRC`, `STREAM`, `BW`, `STREAM_BW`, which inform the bandwidth scanners about a new network consensus document, a new descriptor, a new circuit or stream creation as well as bandwidth information.

The scanners then take a slice of up to 50 relays determined by percentiles, starting at a low percentile and going upwards. Relays that do not possess the FAST or RUNNING flag, are not taken into account for the measurements.
The scanners then start creating two-hop circuits through the Tor network to download a large file, ranging from 128 KB for the slowest relays to 8 MB for the fastest according to the specification, or from 32 KB to 4 MB according to the source code. The rules that apply for the the circuit path selections are as follows :

- relays need a minimum consensus bandwidth of 1
- nodes with the lowest measurement count are prioritized
- relays in the 2 hop circuit come from a different /16 subnet
- entry and exit relays must be fast
- exit relays must permit exit to port 443
- entry relays must not permit exit to port 443

For each slice, the scanner continues creating circuits and downloading through them until each relay in the slice has had at least 5 measurement attempts. In order to take into account network failures, the number of successful downloads should be at least 65% of all possible downloads, which is $\frac{5*nb-relays}{2}$ with $nb-relays$, the number of relays in the current slice.

The measured bandwidth for each measurement is the number of bytes divided by the time delta marked by the STREAM SUCCEEDED event, which indicates that a stream has correctly been attached to a circuit and is functional, and the STREAM CLOSED event, which indicates that the stream has been closed and is no longer functional.

Once an entire slice has been measured, the scanners write out the following information to disk: the slice number, the Unix time stamp when the file was written, and then for each node:

- the fingerprint (a hexadecimal representation of the unique identifier for the relay in the network)

- the nickname (a string up to 20 characters)
- the stream bandwidth, the average of all streams
- the filtered bandwidth, the average of all streams that had a result higher than the stream bandwidth, to exclude possible slow results created by the other node in the circuit
- the descriptor bandwidth, the bandwidth as self-advertised by the relay
- the network status bandwidth, the bandwidth as it is in the consensus

This information is aggregated in the next step to create the bandwidth consensus weight, which is explained in the next section.

1.5.2 Result Aggregation

Once an hour the results provided by the bandwidth scanners are aggregated in order to provide the bandwidth weights for the consensus votes. In order to do so the most recent measurements for each node are taken into account, files being older then 15 days being automatically excluded. There are two methods to aggregate the results, with or without proportional-integral-derivative controller (PID controller).

The first method, without PID controller, is relatively simple. It takes the filtered bandwidth and divides it by the filtered average network bandwidth, then multiplies that ratio by the descriptor bandwidth, which is the router's self reported bandwidth.
This method is a simple way of adjusting the descriptor bandwidth to the relative bandwidth the relay has in the network and to scale it with the load it currently already receives.
When a Tor instance is first started, it creates 4 circuits over the Tor network and sends 125 KB through each of them and measures the largest burst in a 10 second period[12]. This value is then used as self reported bandwidth.

In their votes the authorities have a list of parameters that can be used to control the behavior of the PID controller, one of which is bwauthpid=1 which is set on all 9 authority directories and indicates that all bandwidth scanners use the second method to aggregate the results.

The second method uses a PID controller. The goal of the PID controller is to achieve a perfect load balancing on the Tor network, meaning that a client should receive the same bandwidth no matter which path he chooses on the network. Each deviation from the perfect state can be viewed as being an error. The new consensus bandwidth weight is then calculated as follows:

Data: Input : previous bandwidth weight **bw** - the weight for the node found in the previous consensus
Kp - the proportional gain
Ki - the integral gain
Kd - the derivative gain
Result: consensus bandwidth weight
PID error e(t) = (filtered_measured_bandwidth - network_filtered_average_bandwidth)/network_filtered_average_bandwidth)

weight = bw + bw*Kp*e(t)
+ bw*Ki*integral(e(t))
+ bw*Kd*derivative(e(t))

Algorithm 2: The computation of the new consensus bandwidth weight from [16]

The algorithm follows the standard PID form, meaning that Ki and Kd are proportional to Kp. By default Kp is equal to 1, but the value can be modified by the consensus. The group consisting of nodes which are both Guard and Exit nodes use a different values which is: Kp *(1.0 - the bandwidth weight of the Guard+Exit node group).
Ki is calculated as $\frac{Kp}{integraltime}$ where the integral time is chosen by the developers to be 5 feedback intervals; Kd is equal to $Kp * derivativetime$, where the derivative time is half a feedback interval.
A feedback interval is the time which is believed to be needed for the network to have accepted and adapted to the new bandwidth weights. For Guard nodes, the feedback interval is set to two weeks, for all other nodes it is 4 hours.

There are multiple parameters which can be used by authorities, by voting for certain parameters in their respective voting documents, to tune the PID controller. Some of the possible parameters are:

- Bwauthcircs=1, if present the circuit fail rate may replace the PID error. This makes it possible to reduce load on nodes that fail circuit creation due to high load.
- Bwauthnsbw=1, if present, the previous bandwidth weight **bw** uses the weight form the previous consensus instead of the default descriptor bandwidth. This parameter is currently not set for any of the authorities, which allows for easy bandwidth weight manipulation by changing the descriptor bandwidth.

Additionally the consensus document can overwrite the default values for both the derivative and integral time, but currently do not. By default both values are 0 and thus the values of Ki and Kd become 0 as well. As a result the PID controller method currently converges to the first method presented in this section.

When starting a Tor instance as an authority, the Tor configuration file allows for an option which specifies a path to the file containing the measurements aggregated by the python script. The option is "V3BandwidthsFile" followed by the path of the file. This file, the output

produced by the aggregation, contains the following information:

A Unix timestamp of the file creation and then for each node: the fingerprint, the network consensus bandwidth weight, a timestamp of when it was measured, a timestamp of when it was updated, the PID error e(t), the accumulated error sum, the PID bandwidth which is the last bandwidth value used in the calculation, the delta between the current PID and the previous PID and finally the circuit fail rate.

This information is then used by the authorities to include measurement results in the voting documents.

1.6 Path Selection Algorithm

In this section we shortly explain the Tor path specification as described in [18]. As soon as a Tor instance is started and after it has gathered enough directory information, it begins to build circuits, some of which are used for self testing bandwidth and reachability. A Tor instance will always try to have a few clean circuits, meaning circuits that have yet to be used, at the ready, in order to be able to serve client request as soon as possible.
In order to predict which circuits will be needed, the Tor instance keeps track of each exit port during the last hours and tries to have two clean fast exit circuits for each port up to a maximum of 12 circuits. Circuits may be adequate for multiple ports. Additionally Tor always tries to maintain a fast clean exit circuit to port 80 as well as two fast, clean, stable internal circuits.

The restrictions for path building are as follows:

- no related router in the same path, meaning no router twice, no router of the same family, no two routers in the same /16 subnet
- no non-valid or non-running routers, except for Middle nodes or Rendezvous Points where non-running routers are allowed
- the first node must be a Guard node
- fast/stable circuits only contain routers with the fast/stable flag
- each node is selected proportionally to its consensus bandwidth weight
- Circuits are 3 hop circuits.

A family of routers is an optional Tor parameter, that allows people, which run multiple Tor relays to mark all the routers they administrate as related.
It is important to note that using the Tor control protocol it is possible to build any arbitrary circuits using the EXTENDCIRCUIT command, which is how the bandwidth scanners are able to build two-hop circuits which do not necessarily follow the same rules. To build a 3 hop circuit the command : "EXTENDCIRCUIT 0 fingerprint1,fingerprint2,fingerprint3" can be used. "0" signifies that a new circuit should be created. If the user wants to extend an existing circuit, it can be replaced by the circuit number. Fingerprints 1 to 3 are the fingerprints of the relays through which the circuit will go fingerprint1 being the first and fingerprint3 being the last. If the EXTENDCIRCUIT command was correct, the Tor instance will respond with "250 EXTENDED circuitnumber", where circuitnumber is the number assigned to the circuit.

The selection of nodes is based on the network bandwidth weight as measured and calculated in the previous section. The chance, of being selected is the ratio defined by the network bandwidth fraction the node makes up multiplied by the weight for that particular group. For instance a node with Guard and Exit flag and bandwidth weight bw will have a $\frac{bw}{summed\_network\_bw\_for\_Guard\_position} * Wgd\%$ chance to be chosen as a Guard node by any client and a $\frac{bw}{summed\_network\_bw\_for\_Exit\_position} * Wed\%$ chance to be chosen as an Exit node. Wgd and Wed are weights for Guard and Exit flagged nodes in the Guard position and in the Exit position respectively. All weights for nodes are listed in [18] section 2.2 and the calculation of these weights can be found in [20] section 3.8.3.
The values for weights can be found at the end of each consensus document

During the path selection process, the Exit node is the first node to be chosen. By looking up the exit policy it is easy to know if an Exit relay will support a given IP address. If the IP address is unknown, Tor tries to find an exit that supports the given port number.

Tor also saves circuit buildtimes in order to learn when to give up on a pending circuit, a circuit that has begun building, but has not yet been finished. The default timeout is 60 seconds, meaning the client's OP gives up on a circuit if it has been building for more then 60 seconds. Additionally Tor measures connectivity in order to account for network failures and network connectivity issues. Tor assumes a connectivity loss if no more traffic is received since circuit creation and stops counting timeouts. If in the last 20 circuit creations 18 or more timed out, the timeout is reset to 60 seconds, or doubles if it already was at 60 seconds or above.

Chapter 2

Tracking Detection on Tor Hidden Services

In [7] authors demonstrated that it is possible for a low-resource attacker to control the responsible hidden service directories of a hidden service. As a result the attacker is able to make the hidden service unreachable for clients or is able to track the usage and activity of the hidden service. This section presents a method by which we can indicate with different levels of confidence whether responsible HSDir's of a hidden service were controlled by a malicious/curious entity, as well as an implementation of the presented method. In particular we introduce several indicators which can be used to reveal non-random behaviour; we use this indicators to automatically detect tracking attacks on Tor hidden services. In addition we develop a tool to visually represent potential anomalies.

2.1 Automated Detection

2.1.1 Method Overview

The current implementation of Tor allows one to control responsible hidden service directories of a hidden service. This enables tracking of clients requests. In this section we show that such tracking can be identified using statistical analysis of the consensus history.
Before proceeding with the analysis we recall the details of the Tor Hidden Services protocol. For clients to be able to connect, a hidden service announces its existence and provides contact information; every 24 hours it calculates two new service descriptors and for each service descriptor 3 *responsible hidden service directories* are designated from the set of Tor relays with HSDir flag. The descriptors are then uploaded to the responsible hidden service directories. We call an interval between two consecutive descriptor uploads as "time period". In order for a relay to obtain an HSDir flag it needs to be online for at least 25 hours.
A relay with an HSDir flag becomes a responsible HSDir if its fingerprint (the SHA-1 digest of its public key), is one of the 3 fingerprints that follow the hidden service descriptor's ID. A hidden service descriptor contains all the information necessary to establish a connection to the hidden service.
In order to detect the attack we identify relays which differentiate themselves from the norm with the following indicators :

1. We consider the number of times a relay was responsible for a Hidden Service during the analyzed time period. At any given time period, it is possible to calculate the probability that a relay becomes a responsible hidden service directory for a specific hidden service. If we assume that N_{hsdir} is the number of HSDir's present in the consensus,

that the number is constant and that each responsible HSDir always stays active during its 24 hour period, we can calculate the probability of any HSDir being selected as the responsible HSDir as $p = 6/N_{hsdir}$.
Since we then have a constant probability and independent trials, we can express the number of time periods a Tor relay will be chosen as a binomial distribution with $\mu = np$ and with standard deviation $\sigma = \sqrt{np(1-p)}$. We then can assume that a relay is suspicious, if it exceeds $\mu + 3\sigma$ time periods. We can of course do the same calculation with the number of hours spent as a responsible hidden service directory by simply multiplying the time periods by 24.

2. We look out for close distances by defining a ratio $\frac{d}{avgd}$ with

   - d the distance between two fingerprints, or a fingerprint and a descriptor-id, calculated as: $fingerprint_{n+1} - fingerprint_n$ or $descriptorid$, where fingerprints are expressed as decimal values and arranged in the sorted (ascending) circular list.
   - $avgd$ the average distance between HSDir's for the consensus document.

   The distance is considered close if the ratio is more than 1000.

3. We look out for activity in the near past which could help differentiate a relay from others. In the 3 days before a relay becomes HSDir we look for: fingerprint changes, recent acquiry of the HSDir flag and recent appearance in the consensus.

4. We consider the amount of times a relay changed its fingerprint. Some changes are to be expected but a high amount of changes is unusual.

5. Finally we try to look if a server or a group of servers were responsible HSDir for consecutive time periods.

2.1.2 Implementation Details

We now describe in more detail the implementation. The possible input for the program is :

- an onion address in the format z or z.onion, where z is the permanent-id of the hidden service
- or, the path to a file containing onion addresses in the same format
- the date the user wants to start analyzing the consensus
- the date the user wants to end analyzing the consensus
- a tuple that dictates which information is important enough to be given as an output
- the path to the top level directory containing the consensus (currently hard coded)

The program outputs a number of files. The first file contains the main results. For each onion address that has been analyzed, the number of trackers found is given as well as the number of suspicious relays found. Additionally the information requested by the user with the interest-tuple is output. For each hidden service that is analyzed, a separate file is created containing all HSDir's and all aggregated results. Finally, an errorlog containing possible errors such as missing consensus documents and a timelog showing how much time was spend to compute the different parts of the algorithm, are produced.

Described in broad strokes, the last algorithm works as follows:

1. **The network consensus document is loaded into memory.**
   Specifically for each consensus, we create a new entry in a dictionary using the date as a key, with format "%Y-%B-%d %H" as specified by the python strftime function, containing the following information:

   (a) Two dictionaries which both contain the information : ["HSDir"|None, name, fingerprint, hexvalue, date, time, IP-adress, or_port, dir_port] for each relay in the consensus. The first dictionary uses the fingerprint as key, the second relay uses the IP/or_port pair as key.

   This allows checking in O(1) complexity at any given date, whether or not a fingerprint had the same IP/or_port associated with it, while being able to access any relevant information even in cases where a relay changed fingerprint or a relay was transferred to a different physical server.

   (b) A list with both 32 bit encoded and hexadecimal values of fingerprints of relays with HSDir flags

   (c) A float representing the average distance between HSDir's for the consensus document.

2. **The namegroups are loaded into memory.**
   A namegroup is a list of precomputed names that stand in relation to each other. Names are considered similar, if they have an Levenshtein distance of 1 and a length between 3 and 7, or a Levenshtein distance of 2 or less for names of length 7 to 20 (20 is the maximally allowed length in the consensus). If the name is of length 2 or 1 it is ignored for the computation of similarity. Servers with the name *Unnamed* are also currently ignored. The namegroups were computed using the Levensthein python C extension module [10].

3. **The replicas for the time period are calculated**
   As specified in algorithm 1, we calculate both descriptor-ids or replicas for the given hidden service.

4. **Find the responsible hidden service directories**
   Since we have both the descriptor-id and the sorted list of fingerprints of HSDir's for each consensus document we can perform a binary search to find the set of 6 HSDir's that were responsible at that time.

5. **Extract relevant information for each responsible HSDir**
   At this point, we have the consensus documents in an easily accessible format and we know each one of the 6 responsible HSDir's for that specific hour. We can thus check, if some elements have changed from previous consensus documents.
   We look for changes in:

   (a) presence : did the router already exist n hours before ?

   (b) HSDir flag: did the router have a HSDir flag n hours before ?

   (c) fingerprint : did the fingerprint change n hours before ?

   As currently implemented n represents 1 hour, 25 hours and 72 hours. A possible improvement would be to let the user define time intervals, there might be situation where a more detailed control is appropriate.
   If any elements of these occur, the information is saved.

Additionally we check the distance by calculating a ratio $\frac{d}{avg-d}$ where d is the distance as calculated in 2 and *avg-d* the average distance for the consensus document. If the ratio is higher than a factor 1000, the information is saved with the distance and the ratio.

Finally, between consensus documents we keep track of the number of time periods and hours a HSDir was responsible for the same hidden service.

6. **Search for similarities in names in IP addresses**
   For each name that appeared as a responsible hidden service relay, we look whether or not we have related names in the namegroups, and if we do, we look if these related names appeared as well. Ip addresses are considered as related, if the difference between the last two numbers is less then 99. If two IP addresses are similar the two IP's will be shown followed by the keyword IP as shown in the following example :

   ```
   ['149.9.0.57 9001', '149.9.0.60 9001', 'ip'].
   ```

   If names are similar, the two IP/or_ports are shown as well as both names:

   ```
   ['93.114.43.156 10004', '93.114.40.194 10002', 'TheS1gnul', 'Th3Signul']
   ```

7. **Memorize all responsible HSDir's that changed fingerprint**
   If a responsible HSDir changes fingerprint, we save this information. By doing so it is possible to create a list of HSDir's that were responsible for multiple analysed hidden services.

8. **Output the results**
   When all is done, all there is left to do is to output all the information that was gathered, filtered for relevance.
   The algorithm uses two methods to determine relevance, the first method is a scoring system, the second method is a user defined interest tuple.

   The scoring system works as follows:

   - If the HSDir was missing 1 hour before becoming responsible, it is attributed 1 point, less if it was missing 25 and 72 hours before
   - Each fingerprint change is attributed 1 point
   - If the HSDir recently acquired the HSDir flag, it acquires points depending on how recently the flag was aquired
   - Each close distance gives 1 point
   - For the number of hours as responsible HSDir, $\frac{nb\_hours}{48}$ points are aquired
   - For the number of time periods as responsible HSDir, $\frac{nb\_timeperiods}{2}$ points are aquired

If the score is higher than 10, we mark the relay as tracking; if the score is higher than 7,5 we mark the relay as suspicious. The effectiveness of this scoring system will further be discussed in the conclusion.
The second method lets the user define what results he wishes to see. For instance a user may decide to see each HSDir which has switched fingerprints twice and has been responsible for at least 3 time periods, the corresponding routers would then be output in the main result file.
This is just a method to provide a filtered output of the result, all aggregated results for all HSDir's can still be seen in the result file specific to the analysed hidden service.

If the input was a single onion address, point 7 of the algorithm above is skipped. If the input was a file containing multiple onion addresses, points 3 to 7 are repeated for each address in the list.

In retrospect, it would probably have been more correct to declare two IP addresses as being similar if they belonged to the same /16 subnet, which is how Tor avoids possibly related servers for the path finding logic.

2.1.3 Experimental Results

The script was run over 39841 distinct hidden service addresses for 2013. The addresses were provided by request by the authors of [7]. Due to the large memory requirements the script was run over two periods, the first extending from January 1^{st} 2013 to May 31^{st} 2013, and the second from June 1^{st} 2013 to December 31^{st} 2013.

The first time period had a total runtime of 161592 seconds, or one day and 20 hours, 53 minutes and an average calculation time of 4.056 seconds per hidden service.
The second had a runtime of 313270 seconds, or 3 days and 15 hours, 1 minute and an average calculation time of 7.863 seconds per hidden service.

The large discrepancy of runtime results comes from the additional two month period for the second time period, which could not fit into memory, meaning the computational process was slowed down by the memory swapping process.

The first period found a total of 28 tracking relays, two of which are a confirmation of results we will find for "Silk Road" in the next section, and a total of 26 tracked hidden services[1]. The total number of suspicious servers is 1058.
For the second time period we found a total of 22 tracking relays, a total of 19 tracked hidden services, and a total number of 1052 suspicious relays.
We know from the next section, that a tracking took place to "Silk Road" at the end of May and a second in August but the automatic detection only found the attack performed by relays in the first time period. The main reason, why the attack was not automatically detected[2], is that it was performed by 6 separate servers on a single day. The program does store information on related servers, but they are not taken into account by the score system.
Secondly due to the high number of hidden services analyzed we had to account for the impact of a single fingerprint change. Indeed at the end of 2013 the consensus document shows a total of 5757 relays, 2134 of which were HSDir's. Since we analyzed 39841 hidden services

[1] The list of tracked onion addresses can be obtained upon request.

[2] By looking at the aggregated information for HSDir's of "Silk Road" it is quite easy to detect for a human, but the scoring system failed to mark the relays as suspicious.

and since each hidden services always publishes its descriptor to 6 HSDir's, a HSDir is potentially responsible for more then 100 hidden services. It is straightforward to see that a fingerprint change, even if legitimate, can increase suspicion for multiple hidden services, especially if the fingerprint also places the HSDir in close proximity of the descriptor-id. The score system must thus be tolerant of fingerprint changes in order to not have a high number of false positives.

It is possible that some of the results meaningless. Of the 39841 hidden service a small percentage utilized the authentication mechanism of hidden services, meaning that it is possible that a number of replicas were calculated incorrectly for lack of a descriptor-cookie. As a result, all following calculations and result aggregation on these hidden services would be incorrect.

Due to the issues coupled to analyzing a high number of hidden services, we decided to do the same analysis to a restricted amount of popular hidden services, thus also removing inactive hidden services and hidden services which do not present an interesting target to attackers due to low popularity and use. A list of 30 popular hidden services was retrieved from [3], comprised of 3 categories each containing the 10 most popular hidden services[3]. The list of hidden services analyzed can be found in Annexe E.
These hidden services were analysed in two 6 month periods, the first from August 27^{th} 2013 to February 27^{th} 2014, the second from February 27^{th} 2014 to August 27^{th} 2014.

No tracking or suspicious relays were found for these hidden services on the analyzed time period. Manual control of the result showed that none of relays had at the same time a close distance to the descriptor-id and a recently changed fingerprint.

2.2 Visualization of HSDir Anomalies

In this section we implement a tool for visualisation of anomaly indicators described in the previous section. For the sake of clarity we describe the tool based on an example analysis of “Silk Road”. The infamous “Silk Road” v1.0 marketplace is a well known and popular [7] hidden service and thus a likely target for tracking.

For each consensus[4], the responsible HSDir's were extracted. Figure 2.1 shows the information gathered for one of the replicas from "Silk Road". Each orange bar represents a separate consensus document. The length of the bar represents the average distance between fingerprints for the consensus. Also shown by squares is the distance between the descriptor-id and each of its responsible HSDir's fingerprints. In order to distinguish the different HSDir's, the distance between replica and HSDir 1 is represented in a lighter shade than the distance between replica and HSdir 2 and so on.

If the relay changed fingerprint in the last 3 days or was not present in the consensus, the squares are shown in red to highlight potentially suspicious HSDir's. Otherwise the squares are shown in blue.
Note that changes in fingerprints are expected to happen and are not sufficient to justify suspicion, but since they are almost a necessity for a tracking attack[5], they are a good indicator

[3] Onion addresses in this list are filtered and thus may not contain the most popular hidden services
[4] A new consensus should be produced every hour, however some documents are missing from the archive.
[5] An attacker must either change the public key of his relay, or start the relay with the correct public key

to look out for.

As can been seen in figure 2.2, a mouse over will show the name, IP-address and port as well as the average distance of each of the 3 responsible HSDir's[6] that hour.
Figures 2.2 and 2.1 were both done with AM*CHARTS*, a javascript library for chart creation [5].

[6]The graph only shows information for one out of two replicas

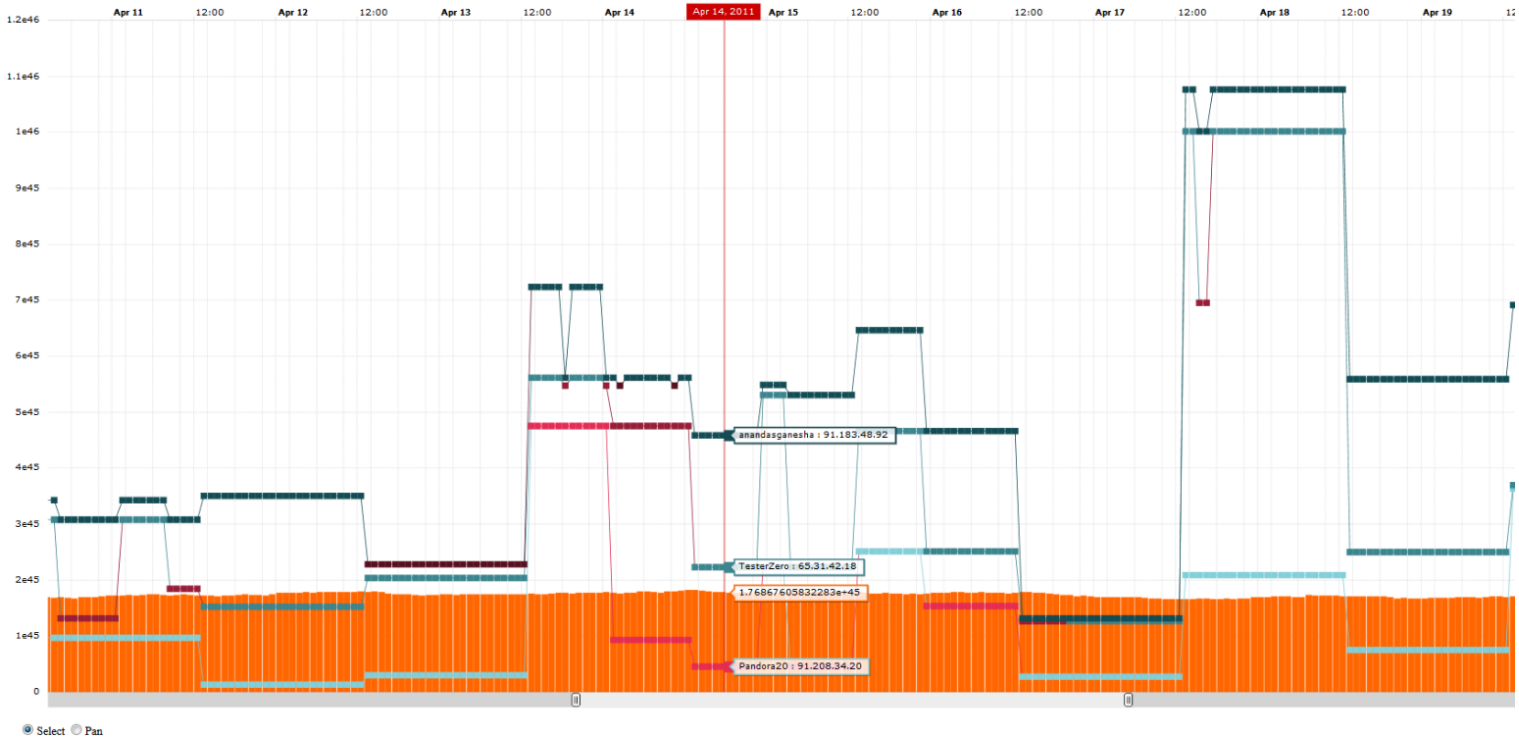

Figure 2.1: An example of visual output from the first iteration

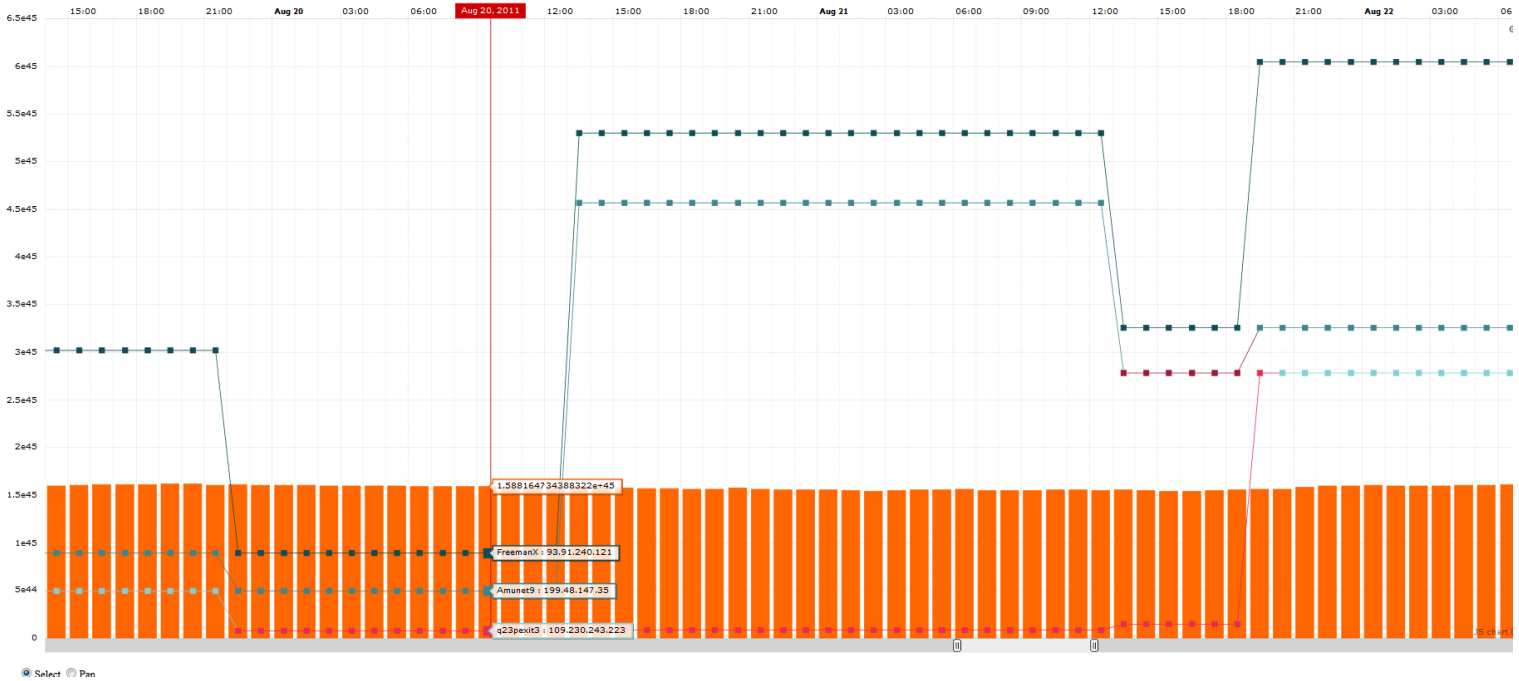

Figure 2.2: A closer look at the first iteration

Chapter 3

Bandwidth Cheating

Originally, Tor was a small network carrying little traffic with few servers and no load balancing. As the network grew, and users with widely varying capacity set up their routers as Tor nodes, it became necessary to implement algorithms to distribute the traffic fairly on all nodes, so that the potential bandwidth throughput of the entire network would be maximized and additionally the overloading of low bandwidth nodes would be prevented.
In the original algorithm the probability to be chosen by a client was proportional to the self reported bandwidth. This opened up a new type of attack [6], in which a low-resource attacker could pretend to have a lot of bandwidth in order to control a high average of paths. An attacker which controls both the Guard node and the Exit node can perform traffic correlation attacks in order to deanonymize parts of the network.

In order to counter these types of attacks, bandwidth scanners were implemented to control and adjust the self-reported bandwidth values with independent measurements, while still providing information in order to strive towards an optimal load distribution.

A lot of research has been conducted to obtain a more optimal load balancing on the Tor network, but to the best of our knowledge no research has been published on attacks against the bandwidth scanners.

This chapter will present a method by which an attacker can inflate its bandwidth consensus weight in order to attract traffic trough his node. We describe the necessary implementation steps for such an attack, and then present our results gathered from 4 servers running as relays on the Tor network for a period of 2 to 3 weeks.

3.1 Method overview

While doing a good job to improve the network load balancing, the current implementation of bandwidth scanners does not provide much security. A careful analysis of the corresponding specification and the source code shows that there are at least two methods which can be used by a low-resource attacker to pretend to have a lot of bandwidth.
In section 1.5.2 we show that: (1) the current values of the PID controller's terms are $K_p = 1, K_i = 0, K_d = 0$; (2) instead of using the bandwidth weight from the previous consensus the self-advertised bandwidth is used in the new weight computation. This effectively results in that the principal part of the consensus weight computation is still the relay's self-advertised bandwidth. By setting its values to a very large number, an attacker is able to get a very large consensus bandwidth weight, while in reality providing very little throughput to clients. This

method is however very noticeable as an attacker needs to set unrealistically large values to self-advertised bandwidth. In addition once the functioning of the PID controller is fixed, the method will not work anymore.
An attacker can instead adopt a more stealthy and harder to fix approach for bandwidth cheating. The attacker can provide unlimited bandwidth for the scanners streams while significantly restricting the speed of the general user's streams. This becomes possible since: (1) the IP addresses of bandwidth scanners are publicly known; (2) the location of downloaded files used for measurement is known; (3) the bandwidth scanners establish two hop circuits. The attacker adopts the following strategy on her relays when relaying traffic;

- If the origin of a stream is one of the Tor authorities, remove bandwidth rate limitation; otherwise throttle traffic as usual
- If the destination of a stream is a URL used by bandwidth scanners, then remove bandwidth rate limitations; otherwise throttle the traffic

This means that bandwidth scanners streams receive high throughput which results in a high consensus weight of the attacker's relays, while the real throughput of the relay as seen by Tor clients remains low.

3.2 Implementation Details

The start of the work on bandwidth cheating was done by testing the existing torflow code[1]. Some tools for Tor network testing exist, such as Chutney[1] and Shadow[2]. However, to make sure to have correct experiment conditions, tests were first performed on a small scale private test Tor network and later on the actual Tor network.

We decided to run exit relays for the experiments since exit relays can: look at traffic to see which websites are visited, gather unencrypted usernames and passwords, reroute traffic to fake websites, and thus constitute a more valuable target for an attacker than Guard or Middle nodes.

The test Tor network consisted of 5 servers, with a total of 10 Tor instances[2]. Among the 10 Tor instances, originally 3 were directory authorities, 2 were clients and 5 were relays. Later, the 2 clients were converted to relays, as there was no need for client emulation, making it 3 directory authorities and 7 relays.

A number of issues were encountered when trying to set up the bandwidth scanners.
Due to the low number of nodes in our test network we only used one scanner which scanned 100 percent of the network instead of the default segmentation with 4 scanners.
A few issues seemed to arise from environment specific conditions[3]. Additionally if a node is responsible for more then 5 percent of the total network bandwidth it is capped to 5 percent, a limitation we had to remove since we only had 10 nodes in our network.

[1] Torflow is a tor project aiming at improving the Tor network; among other things, it contains the bandwidth scanners.

[2] It is possible to run up to 2 Tor servers per IP address, further instances will be ignored by the authorities and not put in the consensus.

[3] Specifically the combination of SQLAlchemy and Elixir did not create the databases correctly, meaning no results would get output. A list of all generated SQL tables by a correct run of the scanners can be found in Annexe D

Since the aggregation calculates a number of weights for Guard nodes and Exit nodes it was necessary to make sure that the network contained both enough guard and Exit nodes. Luckily, it is possible for the authorities in testing networks to give the Guard flag to specific routers without them needing to fulfill the usual conditions. Exit relays only need to have the appropriate exit policy to obtain the Exit flag.

Originally running with an unmodified descriptor, Kalden and Trokel were later modified to have a descriptor bandwidth of 20 MB. The other two relays were restarted in both as exit relays as well also with a descriptor bandwidth of 20 MB (ObservedBW 20.000.000, ReportedBWRate 20.000.000 and ReportedBWBurst 20.000.000).

In order to cheat the bandwidth scanners, one of the first steps was to let the relays return incorrect descriptors, since the self reported bandwidth plays into the consensus bandwidth weight. A descriptor contains information on the routers, such as the fingerprint, the router name and IP address, uptime , bandwidth onion keys etc. A descriptor after modification can be seen for Trokel in Annexe C.

The information we were interested in modifying here is in first place the bandwidth information. The bandwidth information is stored in the line starting with bandwidth followed by 3 integers which represent the self-measured bandwidth, the BandwidthRate and the BandwidthBurst as set in the configuration file. Being able to modify the uptime is also useful, since some flags such as guard are only attributed after the relay has been up and running for some minimum period.

In order to be able to change these values at will, some changes were made to the Tor code and the Tor control port was extended to allow for changes while the Tor instance is running. The uptime value of the server seems to be implemented in such a way that the value which is put in the network consensus votes is the minimum between the reported uptime and the uptime measured on authority side. It is thus not possible to skip the unmeasured period by increasing the uptime.

Another modification that was done refers to the exit policy. To prevent that the university routers are exposed to possible abuse we adopted an exit policy of reject *:*.
However, since we were still interested in the Exit flag, we modified the descriptor to show a normal exit policy allowing for port 80, 443 and 6667, which are the ports for the http, https and irc protocols respectively. At least two of these ports need to be allowed in order to obtain an Exit flag. Additionally we made sure that the two IP addresses used for bandwidth measurements were unblocked. Below is the exit policy displayed for one of our servers.

```
reject 0.0.0.0/8:*
reject 169.254.0.0/16:*
reject 127.0.0.0/8:*
reject 192.168.0.0/16:*
reject 10.0.0.0/8:*
reject 172.16.0.0/12:*
reject 46.4.99.45:*
accept *:80
accept *:443
accept *:6667
```

```
accept 38.229.70.2:*
accept 38.229.72.16:*
reject *:*";
```

In order to restrain the bandwidth used by Tor, the Tor developpers have implemented a bucket system. For both incoming connections and outgoing connections, buckets are filled with the bandwidthburst, and each time a connection reads/writes bytes from an incoming or outgoing connection it subtracts the amount of bytes read/written from the bucket. The bandwidthrate governs how fast the buckets are refilled. If the buckets are empty, all communications are stopped until they have been refilled. A round robin tournament is used in order to distribute the bandwidth fairly under all existing connections.

In order to remove the bandwidth limitations for bandwidth measurements, we added code to detect and mark such measurements. Remember that bandwidth measurements are performed through a 2 hop circuit with the bandwidth scanner on the entry and the website containing file to download on the exit.
Detecting if our server is an Exit node in the 2 hop circuit for a bandwidth measurement is fairly easy. When the exit connection is created, one can look into the payload of the received cell and compare it to the address we know to correspond to the servers which host the files for bandwidth measurements. During our testing of the private Tor network, the two addresses, that we could find are :

- 38.229.72.16
- 38.229.70.2

Additionally, we know that exit connections can only be associated with one circuit, meaning we know that only the bandwidth measurement stream goes through the exit connection. We can thus remove any limitation on the connections associated with circuit.

If the relay is in an entry position, we look for incoming traffic from directory authorities, and we make the assumption that the other real Tor network traffic going through the authorities is minimal and can be ignored. If we see an incoming connection from an authority, we mark the connection as bandwidth connection. If the circuit with the authority is closed, the connection goes back to being limited, to make sure that clients cannot use the connection for other circuits and have unlimited bandwidth.

The bucket system also allows to check on bandwidth consumption. In figure 3.1 it is relatively easy to see the real 50 KB/s bandwidth cap we imposed on the relay. The graph was produced by Atlas [11].

3.3 Experimental Results

For the experiment on the Tor network, a total of 6 relays were used, they can be found in Table 3.1. As can be seen in this table, on the 14th of July at 11 AM 4 servers with the names Cehad, Trokel, Kalden and Triss were started. Each of them had a descriptor set with a BandwidthRate and BandwidthBurst of 50 KB/s, two were exit relays with a restricted exit policy [15], a policy designed to reduce potential abuse. The two others were non exit relays.

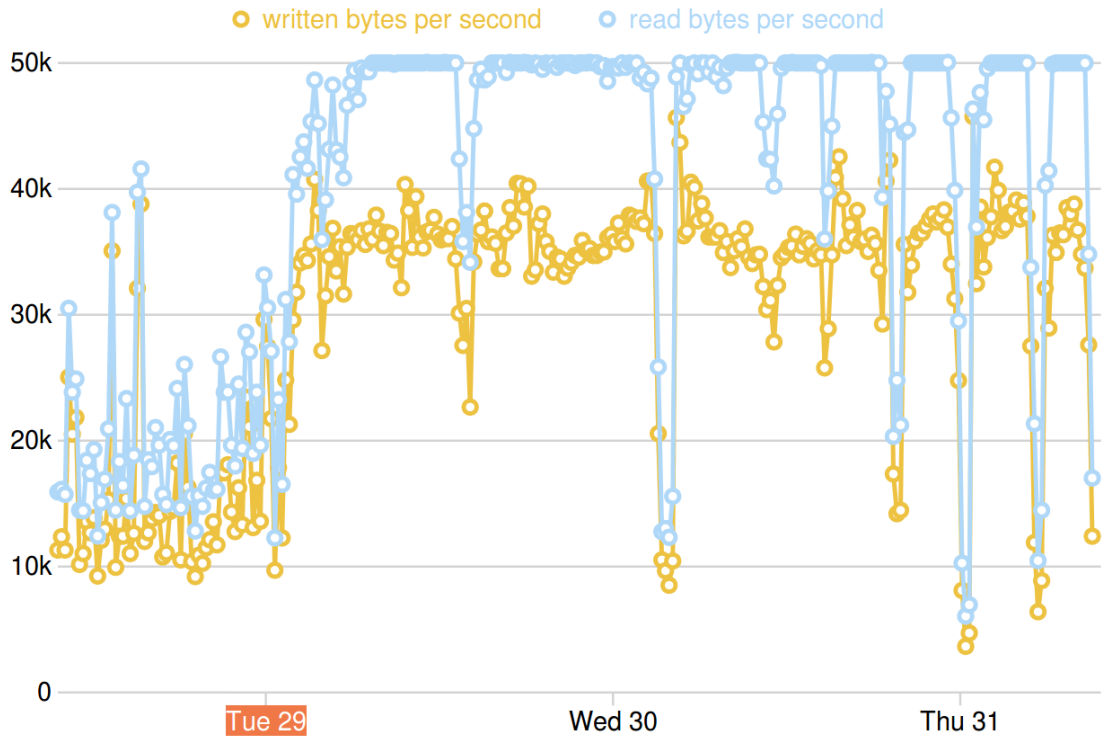

Figure 3.1: Dandelion with visible bandwidth cap at 50 KB/s

Relay Name	Fingerprint	Type	Descriptor	Online
Triss	CA0462A1...	non-exit	unmodified	14Jul-11:00 to 23Jul-13:00
Cehad	0AA7121D...	non-exit	unmodified	14Jul-11:00 to 23Jul-13:00
Kalden	401124BD...	exit	hybrid	14Jul-11:00 to 31Jul-11:00
Trokel	2F6308D8...	exit	hybrid	14Jul-11:00 to 12Aug-12:00
Dandelion	5EDF4156...	exit	modified	23Jul-13:00 to 31Jul-13:00
Gerald	232883EE...	exit	modified	23Jul-13:00 to 10Aug-18:00

Table 3.1: Relays and their uptime that were used for the experiments on the Tor network

We estimate that relays got measured every 4 to 24 hours, with the average tending to one measurement every 12 hours. The rate of measurements is higher for relays with a higher bandwidth consensus weight.

When a relay is started, Tor sets the the consensus bandwidth weight to 20 and adds an unmeasured flag to show that the bandwidth for the relay has yet to be measured. All 4 of the original relays kept their consensus bandwidth weight around 20, when the unmeasured flag was removed; some even dropped lower as for instance Cehad. Figure 3.2 shows the bandwidth consensus weight from the 14th of July at 11 am to the 23th of July at 1pm.

After seeing that the results were stagnant at a low bandwidth consensus weight, the descriptor was changed in order to self report a bandwidth of 20 MB. (The ObservedBW, RePortedBWRate and ReportedBWBurst were all set to 20.000.000). Both exit servers were simply modified, the two non-exit relays were restarted with new public keys and set up as exit relays as well. Results from the descriptor change were pretty much immediate. After 17 hours both relays, that were left running, had their consensus bandwidth weight double. Trokel

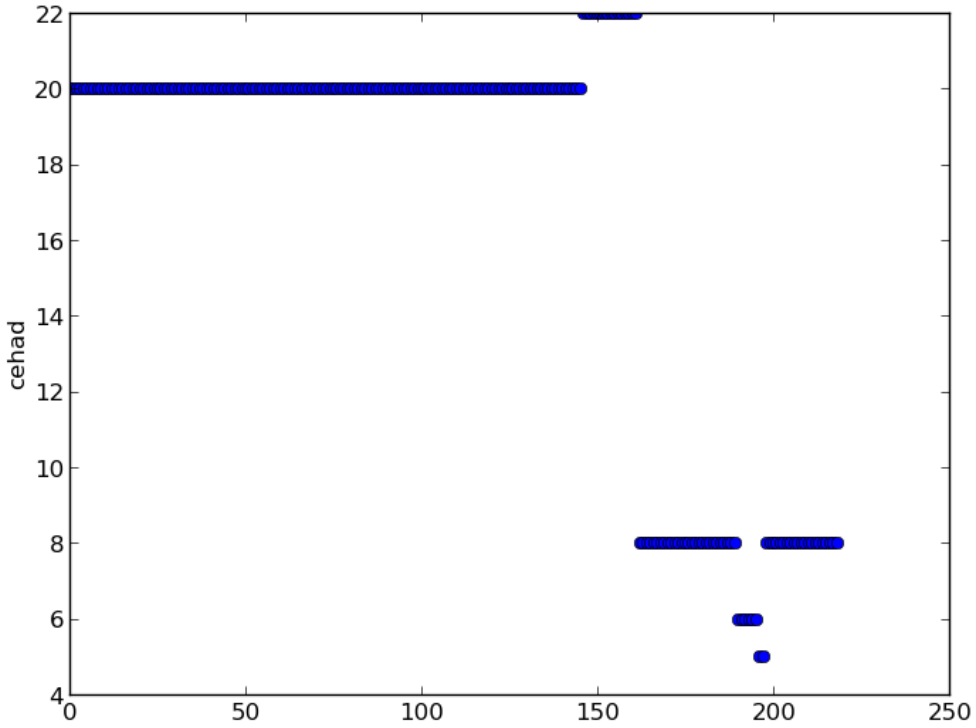

Figure 3.2: Cehad consensus bandwidth weight

jumped to 43 and Kalden jumped to 39. Gerald, one of the servers that was restarted, had the bandwidth consensus weight change from 20 to 1780 after 27 hours.

Since Dandelion and Cehad were still running on the same physical server, we assume that the actual bandwidth measurements remain unchanged from the first tries. This tells us two important factors. The first one is, that the initial placement matters a lot for the bandwidth measurements. If your router is being measured with slow routers, even using filtered bandwidth, it will probably not get an appropriate bandwidth weight, in this case 50 KB/s, even if the actual measurements for the server should be higher then 50 KB/s. (Note: Even if we have no way of looking at the actual measurements results, our own results on the testing Tor network and later results give us no reason to believe that measurements lower then 50 KB/s would occur if not for slow relays in the circuit.)

The server that reached highest bandwidth in the consensus, was the server Dandelion which reached a network consensus bandwidth weight of 77.200 the 2014-07-31-02-00, making it the 21^{st} highest relay in the consensus out of 5.834 active relays.
It can be seen that towards the end the bandwidth consensus weight drops to 0 regularly. The reason for these drops is that Dandelion was not given the Running flag by a majority of directory authorities and thus no longer appeared in the consensus. It is not entirely clear why this was the case; one possible explanation is that with the high traffic Dandelion was receiving due to its high weight, Dandelion was not able to properly treat all incoming connections, making the directory authorities fail to connect to the router.

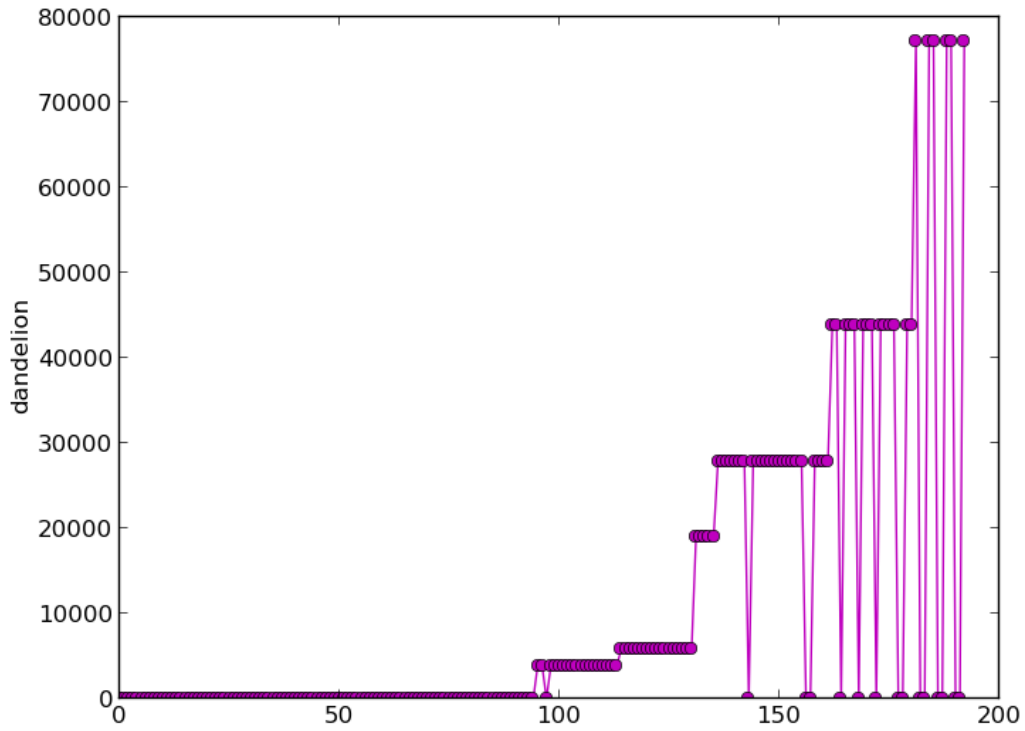

Figure 3.3: Dandelion consensus bandwidth weight

The 9 authority votes on Dandelion are displayed in table 3.2. As can be seen, only 3 servers had a measured bandwidth and the final value found in the consensus is the median of all 3 votes. All the authorities except for tor26 voted the flags Exit Fast Running Valid, tor26 voted only Running and Valid. This made it possible to reach 1 percent in probability to be chosen as an exit relay, as shown in 3.4, a graph was produced by Atlas [11].

Ethical considerations: Since the relays high bandwidth weight brought in a lot of traffic which could not be served properly, we took the relay down as soon as we noticed the high consensus bandwidth weight. It is possible that even higher bandwidth weights could have been achieved had we not stopped the relay.

It is also interesting to look at Trokel in figure 3.5, the relay which had the longest uptime. The two dips to 0 after the initial low consensus bandwidth of 20 were due to the relay being down; the first time it was taken down to make changes to the Tor code, the second time the relay probably crashed due to a special case for which the code we changed was not prepared. It is interesting to note that the consensus bandwidth weight tends to fluctuate a lot, but that an upwards trend can be seen.

Authority	Fingerprint	Vote
Faravahar	EFCBE720...	Bandwidth=10000
gabelmoo	ED03BB61...	Bandwidth=10000 Measured=92300
dizum	E8A9C45E...	Bandwidth=10000
moria1	D586D183...	Bandwidth=10000 Measured=77200
urras	80550987...	Bandwidth=10000
dannenberg	585769C7...	Bandwidth=10000
maatuska	49015F78...	Bandwidth=10000 Measured=25900
turtles	27B6B599...	Bandwidth=10000
tor26	14C131DF...	Bandwidth=10000

Table 3.2: Authority votes on 2014-07-31-02-00 for Dandelion

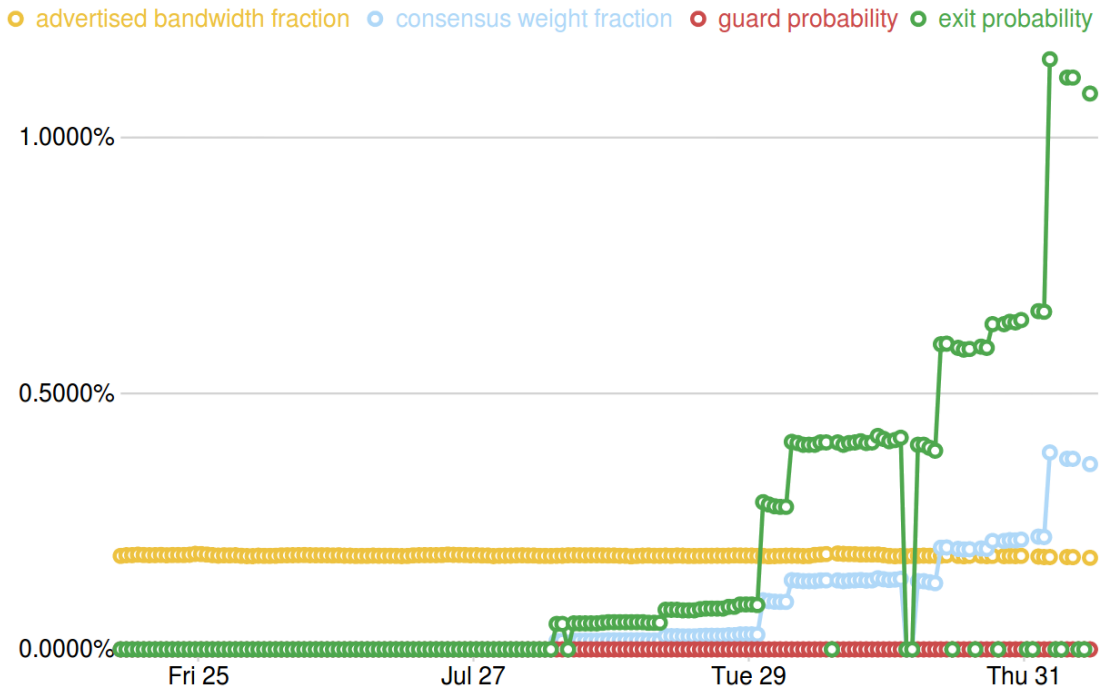

Figure 3.4: Dandelion guard and exit probabilities

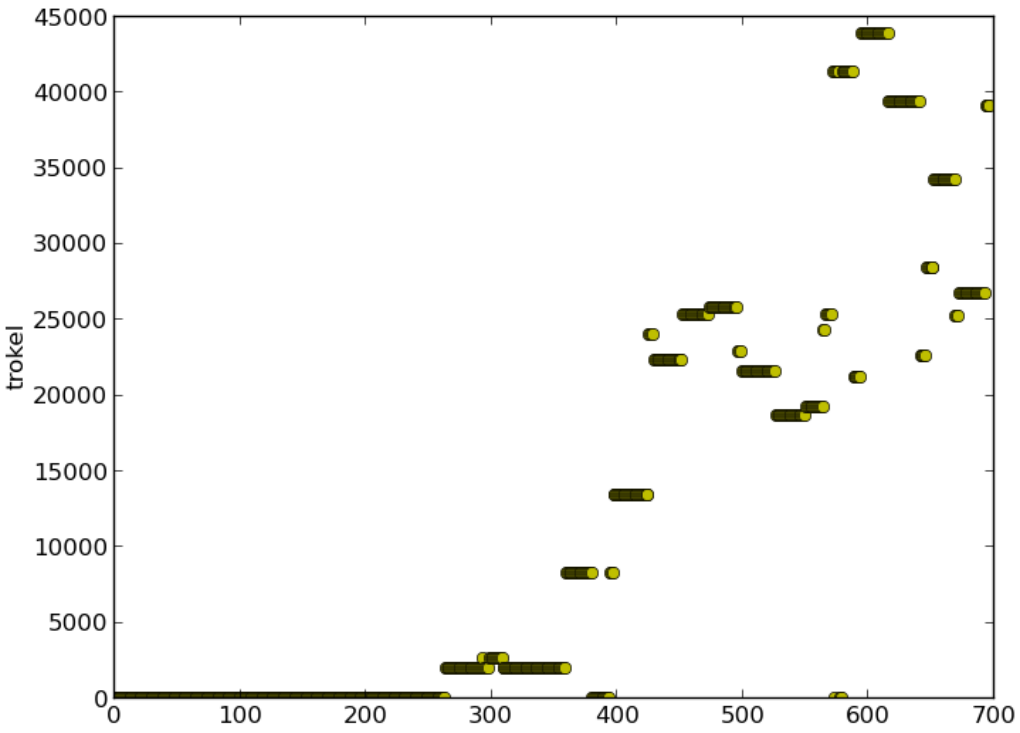

Figure 3.5: Trokel consensus bandwidth weight

Chapter 4

Concluding remarks

This final chapter constitutes a conclusion of both methods presented in this writing, and presents final thoughts on both parts.
This thesis had two main contributions:

- The first contribution consists of a method and implementation of tracking detection. We were able to find a total of 50 tracking relays and 45 tracked hidden services addresses by analyzing 39841 hidden services for the year 2013. The high amount a analyzed hidden services as well as the a small percentage of secret hidden services, increases the potential for false positives. We performed a second analysis of popular hidden service addresses, in the time period from Aug 27^{th} 2013 to Aug 27^{th} 2014, which found no tracking relays.

- The second contribution consists of a method and implementation to artificially increase the bandwidth consensus weight. Using the flaws in the bandwidth measurement system, we were able to position one of our relays amongst the 21^{st} highest relays using a bandwidth of only 50 KB/s.

4.1 Discussion on Tracking of Hidden Services

The point system presented in section 2.2.2, still has a lot of room for improvement. Simple improvements would be to let the algorithm determine the binomial distribution by itself and distribute points weighted by the probability of an event occurring.
A more flexible point distribution for fingerprint in regards to time would also be appropriate. Indeed, two fingerprint changes at random moments in time might represent less information then two fingerprints in rapid succession.

Furthermore the point system should include information on relations between servers and distribute points based on observations for a group of related servers. Indeed for a human it is straightforward to see that 6 relays having a close distance to the replica over a 6 month time period is not particularly suspicious, but 6 relays having a close distance the same day is an almost guaranteed tracking attempt due to the improbability of this occurrence.

Future work could thus extend the scoring system to take into account time and relations between relays as well as a better point distribution based on automated calculation of probabilities.

4.2 Discussion and Countermeasures on Bandwidth Cheating

In the second part of our writing we show that it is possible to reach a high percentage chance to be selected as an exit relay while spending very little bandwidth. In our attack we decide to allow for a minimum bandwidth of 50 KB/s, to show that an attacker could still observe the traffic which is interesting to him, while dropping other connections.

It is also noteworthy that this type of attack could potentially be used to deteriorate the Tor network. By having multiple relays with high bandwidth consensus values, a client would have a high chance of having an almost unusable connection, deterring clients from using the Tor network.

The two main facilitators of the attack are the self-advertised bandwidth, which an attacker can manipulate to change the bandwidth weight. However this could easily be made impossible if the authorities decide to use the old consensus bandwidth weight instead of the self-advertised bandwidth, as the system is already in place. The second facilitator is the fact that an attacker knows which relays perform the bandwidth measurements and which servers store the files used to measure the bandwidth. The second part of the attack could be prevented by making the bandwidth measurements indistinguishable from normal relayed Tor data.

List of Figures

List of Tables

Bibliography

[1] Chutney. https://gitweb.torproject.org/chutney.git. [Online].

[2] Shadow. http://shadow.github.io/. [Online].

[3] Tor Hidden Service Search. https://www.ahmia.fi/stats/viewer. accessed August 27 2014.

[4] Wikipedia. the hidden wiki, July, 2013. http://en.wikipedia.org/wiki/The_Hidden_Wiki.

[5] Amcharts. AMCHARTS. http://www.amcharts.com/. [Online].

[6] Kevin Bauer, Damon McCoy, Dirk Grunwald, Tadayoshi Kohno, and Douglas Sicker. Low-resource routing attacks against tor. In *Proceedings of the 2007 ACM workshop on Privacy in electronic society*, pages 11–20. ACM, 2007.

[7] Alex Biryukov, Ivan Pustogarov, and Ralf-Philipp Weinmann. Content and popularity analysis of tor hidden services. *arXiv preprint arXiv:1308.6768*, 2013.

[8] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: The second-generation onion router. Technical report, DTIC Document, 2004.

[9] Karsten Loesing. *Privacy-enhancing Technologies for Private Services*. PhD thesis, University of Bamberg, May 2009.

[10] David Necas. Python Levenshtein module. https://github.com/miohtama/python-Levenshtein. [Python module].

[11] The Tor Project. Atlas. https://atlas.torproject.org/. [Online].

[12] The Tor Project. Lifecicle of a relay. https://blog.torproject.org/blog/lifecycle-of-a-new-relay. [Online].

[13] The Tor project. Organization. https://www.torproject.org/about/corepeople.html.en. [Online].

[14] The Tor project. Overview. https://www.torproject.org/about/overview.html.en. [Online].

[15] The Tor Project. Restricted access policy. https://trac.torproject.org/projects/tor/wiki/doc/ReducedExitPolicy. [Online].

[16] The Tor Project. Tor Bandwidth specification. https://gitweb.torproject.org/torflow.git/blob/HEAD:/NetworkScanners/BwAuthority/README.spec.txt. [Online].

[17] The Tor project. Tor files. https://collector.torproject.org/formats.html. [Online].

[18] The Tor Project. Tor Path Specification. https://gitweb.torproject.org/torspec.git?a=blob_plain;hb=HEAD;f=path-spec.txt. [Online].

[19] The Tor project. Tor Rendezvous specification document. https://gitweb.torproject.org/torspec.git?a=blob_plain;hb=HEAD;f=rend-spec.txt. [Online].

[20] The Tor project. Tor specification document. https://gitweb.torproject.org/torspec.git/blob/HEAD:/dir-spec.txt. [Online].

[21] The Tor Project. Torflow. https://gitweb.torproject.org/torflow.git.

[22] Andrei Serjantov and Peter Sewell. Passive attack analysis for connection-based anonymity systems. In *Proceedings of ESORICS 2003*, October 2003.

[23] Eric Wustrow, Scott Wolchok, Ian Goldberg, and J. Alex Halderman. Telex: Anticensorship in the network infrastructure. In *Proceedings of the 20th USENIX Security Symposium*, August 2011.

Appendix A

Additional Resources

Tor and its side projects gather a lot of publicly available information on relays, the network state etc. Some of the related websites are presented here:

First , the current consensus documents can be downloaded directly from the authorities on the page
`http://<hostname>/tor/status-vote/current/consensus`
where <hostname> is the IP address of the authority.

Additionally it is possible to see the current uploaded descriptor of a relay by accessing
`http://<hostname>/tor/status-vote/current/<fp>`
where <hostname> is again the authorities IP address and <fp> is the fingerprint of the relay.

Information on relays can also be found on
`https://atlas.torproject.org/`
along with graphs on the bandwidth use and the probabilities of being chosen. To look for a specific relay on can either use the search function on the website or access
`https://atlas.torproject.org/#details/<fp>`

A list of all running relays can also be found under
`http://torstatus.blutmagie.de/index.php?SR=Bandwidth&SO=Desc`
which shows flags bandwidth and hostnames.
Finally the list of all authority votes for each consensus and each relay can be found under
`https://consensus-health.torproject.org/`

Appendix B

Results Format

The main output of the program is the result overview file. The result overview file contains for each onion address, the list of IP/or_port which were considered tracking or suspicious. Additionally it contains the list of all routers filtered by the interest tuple in the following format:

Field	Values	Explanation
server with ip and port	176.31.119.209 9000	self-explanatory
missing	'2013-May-22 13, 72	the server was HSDir on the given date but was missing from the consensus n hours before
fpchange	2013-May-30 13', 72, 'GsTaUbm...', 'uBtDwBX...'	n hours before becoming HSDir the server change fingerprint from x to y
hsdirget	2013-May-22 13', 25	did the server have the HSDir flag n hours before becoming responsible HSDir. if present then yes
distances	'2013-May-22 13', 10000, 17556, 5631602...	date, the broken limit, the calculated distance and the fingerprint in decimal
hours	96	accumulated hours as responsible HSDir
timeperiods	4	accumulated time periods a responsible HSDir's where a time period is the 24 hour period between HSDir rotation

The file dedicated to a hidden service displays every ip/port which was a responsible HSDir for the hidden service in the same format. At the end of the file a list is shown of all related names and IP addresses found for the same hidden service.

Appendix C

Modified Descriptor for Trokel

```
router Trokel 88.198.15.47 5003 0 0
platform Tor 0.2.5.3-alpha-dev on Linux
protocols Link 1 2 Circuit 1
published 2014-07-30 05:18:48
fingerprint 2F63 08D8 FF63 AF7C DF43 DEE6 E655 A11B 73F2 02DD
uptime 1313543
bandwidth 20000000 20000000 20000000
extra-info-digest 0912E78DF5C78EA417E950483070368418098555
onion-key
-----BEGIN RSA PUBLIC KEY-----
MIGJAoGBAKOXQUVAlOOxg+hk6xOKWNmST092tQZTXug2Jl1e6/Zy4S3Roe6Pn655
IP26VidKTbTGDULOOo4QCVoKxMh2ORolgE8jGpKdteLhiEfxxYQkZLYL3X8dEarS
MZ1g1TdWc4Ut3FgItTM6xa4zpugtL+2VoQMmFSvdb1QAqg3V+1N5AgMBAAE=
-----END RSA PUBLIC KEY-----
signing-key
-----BEGIN RSA PUBLIC KEY-----
MIGJAoGBALhuJBw/EAeVOW7h2yfv5aWYn1Dz7QrtNea/XJ5kdeoGqvfHAa5t6UHC
9lVkTnHTp4+YvQIQHQG+pkOURqPuYKj78/VjshYzEnfm9RjQQ8KtqONiexM+1I/i
cJiKaiU54FGWE2TDp5/J8quIva8ciwZNcKDi3e8gKLHb2uGPEly1AgMBAAE=
-----END RSA PUBLIC KEY-----
hidden-service-dir
contact fabrice.thill.001ATstudent.uni.lu, ivan.pustogarovATuni.lu
ntor-onion-key p3zs3znIgk0PWR6/uE2/CEPQoekCvHgybjBcEn2hxAY=
reject 0.0.0.0/8:*
reject 169.254.0.0/16:*
reject 127.0.0.0/8:*
reject 192.168.0.0/16:*
reject 10.0.0.0/8:*
reject 172.16.0.0/12:*
reject 88.198.15.47:*
accept *:80
accept *:443
accept *:6667
accept 38.229.70.2:*
accept 38.229.72.16:*
reject *:*
router-signature
```

```
-----BEGIN SIGNATURE-----
CBMgVvhe8daEjwKzCMU2WSGhIla/m6Xp0VtZo8VunSBgMcH1y++xwumJt7nXGCc1
hJB9zysxKoDJJWig/sdvJk6ydpalnpi+4B/OEr7RIWfsocRmd+G4Aqyks0I7QuPl
5WNp0Ce5aLGnxi1m/dVYmi1WKJm2kCtDHjvOKSGcqrA=
-----END SIGNATURE-----
```

Appendix D

Bandwidth Tables Format

```
tablecircuitcircuit14CREATE TABLE circuit (
        id INTEGER NOT NULL,
        circ_id INTEGER,
        launch_time FLOAT,
        last_extend FLOAT,
        row_type VARCHAR(40),
        fail_reason TEXT,
        fail_time FLOAT,
        built_time FLOAT,
        tot_delta FLOAT,
        destroy_reason TEXT,
        destroy_time FLOAT,
        closed_time FLOAT,
        PRIMARY KEY (id)
)
indexix_circuit_circ_idcircuit59CREATE INDEX ix_circuit_circ_id ON
circuit (circ_id)
tablerouterrouter68CREATE TABLE router (
        idhex CHAR(40) NOT NULL,
        orhash CHAR(27),
        published DATETIME,
        nickname TEXT,
        os TEXT,
        rate_limited BOOLEAN,
        guard BOOLEAN,
        exit BOOLEAN,
        stable BOOLEAN,
        v2dir BOOLEAN,
        v3dir BOOLEAN,
        hsdir BOOLEAN,
        bw INTEGER,
        version INTEGER,
        PRIMARY KEY (idhex),
        CHECK (rate_limited IN (0, 1)),
        CHECK (guard IN (0, 1)),
        CHECK (exit IN (0, 1)),
```

```
        CHECK (stable IN (0, 1)),
        CHECK (v2dir IN (0, 1)),
        CHECK (v3dir IN (0, 1)),
        CHECK (hsdir IN (0, 1))
)
indexsqlite_autoindex_router_1router69
indexix_router_idhexrouter75CREATE INDEX ix_router_idhex ON router (idhex
)
tablerouter_circuits__circuit_routersrouter_circuits__circuit_routers
117CREATE TABLE router_circuits__circuit_routers (
        router_idhex CHAR(40) NOT NULL,
        circuit_id INTEGER NOT NULL,
        PRIMARY KEY (router_idhex, circuit_id),
        CONSTRAINT router_circuits_fk FOREIGN KEY(router_idhex) REFERENCES
         router (idhex),
        CONSTRAINT circuit_routers_fk FOREIGN KEY(circuit_id) REFERENCES
        circuit (id)
)
index
sqlite_autoindex_router_circuits__circuit_routers_1router_circuits__circuit_routers
121
tablestreamstream141CREATE TABLE stream (
        id INTEGER NOT NULL,
        tgt_host TEXT,
        tgt_port INTEGER,
        circuit_id INTEGER,
        ignored BOOLEAN,
        strm_id INTEGER,
        start_time FLOAT,
        tot_read_bytes INTEGER,
        tot_write_bytes INTEGER,
        init_status TEXT,
        close_reason TEXT,
        row_type VARCHAR(40),
        fail_reason TEXT,
        fail_time FLOAT,
        end_time FLOAT,
        read_bandwidth FLOAT,
        write_bandwidth FLOAT,
        PRIMARY KEY (id),
        CONSTRAINT stream_circuit_id_fk FOREIGN KEY(circuit_id) REFERENCES
         circuit (id),
        CHECK (ignored IN (0, 1))
)
indexix_stream_circuit_idstream172CREATE INDEX ix_stream_circuit_id ON
stream (circuit_id)
indexix_stream_strm_idstream174CREATE INDEX ix_stream_strm_id ON stream (
strm_id)
tablebwhistorybwhistory175CREATE TABLE bwhistory (
        id INTEGER NOT NULL,
```

```
        router_idhex CHAR(40),
        bw INTEGER,
        desc_bw INTEGER,
        rank INTEGER,
        pub_time DATETIME,
        PRIMARY KEY (id),
        CONSTRAINT bwhistory_router_idhex_fk FOREIGN KEY(router_idhex)
        REFERENCES router (idhex)
)
indexix_bwhistory_router_idhexbwhistory179CREATE INDEX
ix_bwhistory_router_idhex ON bwhistory (router_idhex)
tableextensionextension180CREATE TABLE extension (
        id INTEGER NOT NULL,
        circ_id INTEGER,
        from_node_idhex CHAR(40),
        to_node_idhex CHAR(40),
        hop INTEGER,
        time FLOAT,
        delta FLOAT,
        row_type VARCHAR(40),
        reason TEXT,
        PRIMARY KEY (id),
        CONSTRAINT extension_circ_id_fk FOREIGN KEY(circ_id) REFERENCES
        circuit (id),
        CONSTRAINT extension_from_node_idhex_fk FOREIGN KEY(
        from_node_idhex) REFERENCES router (idhex),
        CONSTRAINT extension_to_node_idhex_fk FOREIGN KEY(to_node_idhex)
        REFERENCES router (idhex)
)
indexix_extension_to_node_idhexextension182CREATE INDEX
ix_extension_to_node_idhex ON extension (to_node_idhex)
indexix_extension_circ_idextension189CREATE INDEX ix_extension_circ_id ON
 extension (circ_id)
indexix_extension_from_node_idhexextension195CREATE INDEX
ix_extension_from_node_idhex ON extension (from_node_idhex)
tablerouter_streams__streamrouter_streams__stream198CREATE TABLE
router_streams__stream (
        router_idhex CHAR(40) NOT NULL,
        stream_id INTEGER NOT NULL,
        PRIMARY KEY (router_idhex, stream_id),
        CONSTRAINT router_streams_fk FOREIGN KEY(router_idhex) REFERENCES
        router (idhex),
        CONSTRAINT router_streams_inverse_fk FOREIGN KEY(stream_id)
        REFERENCES stream (id)
)
indexsqlite_autoindex_router_streams__stream_1router_streams__stream212
table
stream_detached_circuits__circuit_detached_streamsstream_detached_circuits__circuit_detac
239CREATE TABLE stream_detached_circuits__circuit_detached_streams (
        circuit_id INTEGER NOT NULL,
```

```
        stream_id INTEGER NOT NULL,
        PRIMARY KEY (circuit_id, stream_id),
        CONSTRAINT circuit_detached_streams_fk FOREIGN KEY(circuit_id)
        REFERENCES circuit (id),
        CONSTRAINT stream_detached_circuits_fk FOREIGN KEY(stream_id)
        REFERENCES stream (id)
)
index
sqlite_autoindex_stream_detached_circuits__circuit_detached_streams_1stream_detached_circui
356
tablerouter_detached_streams__streamrouter\_detached\_streams\_\_stream
474CREATE TABLE router_detached_streams__stream (
        router_idhex CHAR(40) NOT NULL,
        stream_id INTEGER NOT NULL,
        PRIMARY KEY (router_idhex, stream_id),
        CONSTRAINT router_detached_streams_fk FOREIGN KEY(router_idhex)
        REFERENCES router (idhex),
        CONSTRAINT router_detached_streams_inverse_fk FOREIGN KEY(
        stream_id) REFERENCES stream (id)
)
index
sqlite_autoindex_router_detached_streams__stream_1router_detached_streams__stream
501
tablerouterstatsrouterstats9CREATE TABLE routerstats (
        id INTEGER NOT NULL,
        router_idhex CHAR(40),
        min_rank INTEGER,
        avg_rank FLOAT,
        max_rank INTEGER,
        avg_bw FLOAT,
        avg_desc_bw FLOAT,
        percentile FLOAT,
        circ_fail_to FLOAT,
        circ_fail_from FLOAT,
        circ_try_to FLOAT,
        circ_try_from FLOAT,
        circ_from_rate FLOAT,
        circ_to_rate FLOAT,
        circ_bi_rate FLOAT,
        circ_to_ratio FLOAT,
        circ_from_ratio FLOAT,
        circ_bi_ratio FLOAT,
        avg_first_ext FLOAT,
        ext_ratio FLOAT,
        strm_try INTEGER,
        strm_closed INTEGER,
        sbw FLOAT,
        sbw_dev FLOAT,
        sbw_ratio FLOAT,
        filt_sbw FLOAT,
```

```
        filt_sbw_ratio FLOAT,
        PRIMARY KEY (id),
        CONSTRAINT routerstats_router_idhex_fk FOREIGN KEY(router_idhex)
        REFERENCES router (idhex)
)
indexix_routerstats_router_idhexrouterstats2377CREATE INDEX
ix_routerstats_router_idhex ON routerstats (router_idhex)
```

Appendix E

List of used Hidden Service Addresses

The following list of popular hidden services was retrieved from [3] on August 27^{th} 2014. The list is split in the 3 following categories of popularity:

1. by total search results clicks
    - nope7beergoa64ih.onion
    - torbookdjwhjnju4.onion
    - qm3monarchzifkwa.onion
    - zqiirytam276uogb.onion
    - 33lwkzt672innsj6.onion
    - kpvz7kpmcmne52qf.onion
    - hwiki2tzj277eepp.onion
    - jjvxrbpckwpz3kwu.onion
    - 3eocs2e3bwdd4zdv.onion
    - s6cco2jylmxqcdeh.onion
2. by Tor2web average visits
    - pinkmethuylnenlz.onion
    - kpai7ycr7jxqkilp.onion
    - torbookdjwhjnju4.onion
    - h3vf5leilsvjiqlx.onion
    - zxjfcvfvhqfqsrpz.onion
    - rgjqiprwku7axxn3.onion
    - juvatztgkapzrp2o.onion
    - girlshtjireiazwm.onion
    - jf64ppeloydrqbfp.onion
    - bviurfdvzicwbbkv.onion
3. by public WWW backlinks
    - strngbxhwyuu37a3.onion
    - kpvz7ki2v5agwt35.onion

- silkroad6ownowfk.onion
- silkroad5v7dywlc.onion
- flibustahezeous3.onion
- am4wuhz3zifexz5u.onion
- 3g2upl4pq6kufc4m.onion
- xmh57jrzrnw6insl.onion
- xiwayy2kn32bo3ko.onion
- grams7enufi7jmdl.onion