Tominator: A Tor fingerprinting suite.

Or how the Tor-network might get a surprise attack from the future.
“"I'll be back” – The Terminator

PETTER SALMINEN

Master’s Degree Project
Stockholm, Sweden December 28, 2014

XR-EE-LCN 2014:012
The king of high-secure, low-latency anonymity

ABSTRACT

Tor is a very popular anonymisation software and network. For which we created Torminator, a fingerprinting suite written in the Java programming language. Fingerprinting is an attack type applicable to Tor utilising side-channel information from the network packets. With side-channel data, we can analytically access information that purportedly been hidden by design by Tor. Because Tor is a low-latency, low-overhead by design, it will leak communication patterns with intermediate (thus total) communication size. In our case this may able us figure out to which site/service the Tor user is using. This means that anyone with access to user’s traffic can use the fingerprinting attack to partly compromise the provided anonymity. By investigating such attacks, it may help us to better understand how to withstand and resist attacks from powerful adversaries such as state agencies.

Torminator automatises the process for gathering fingerprints. It uses the official Tor Browser through its GUI to enter websites to recreate the real world scenario. This gives us real and reliable fingerprints without having to employ a human to do anything, as Torminator simulates user interaction on Tor Browser for us. We can also give Torminator a list of websites to fingerprint, making it easy to generate lots of fingerprints for a great number given sites.

A contribution of Torminator, is that we improved on the previous de facto standard of the fingerprints collected from the available tools from previous works. We have gathered fingerprints and have now a dataset of 65792 fingerprints. Fingerprints like these can be used with machine learning techniques to teach a machine to recognise web-pages by reading the packet size and directions saved in the fingerprint files.

1 Quote from a leaked top-secret NSA document, acknowledging the fundamental importance of Tor as a software.
ACKNOWLEDGEMENTS

This has been written in first person, as a more personal 'thank you' to all listed persons and organisations.

I want to start off and thank my professor Panagiotis Papadimitratos for opening the position and chance for me to explore this area out of the idea and interest spawned.

Secondly, I want to thank SUNET for sparking the idea and interest in fingerprinting Tor. And of course hosting the university networks on which my fingerprinting operations has been operating on.

Thirdly, I want to thank DFRI, for hosting Tor nodes to circumvent censorship and its members voluntary efforts of helping the Tor network.

And last but not least, I want to thank the Networked Systems Security Group at the School of Electrical Engineering at KTH for lending me a computer and access to the thesis room in which I have spent countless hours on this project.
CONTENTS

1 Introduction .............................................. 1
  1.1 Personal motivation and purpose ................. 2
  1.2 Previous work ..................................... 3

2 Tor .................................................. 5
  2.1 Tor Nodes ........................................ 6
    2.1.1 Classification of Tor nodes ............... 8
    2.1.2 Node flags .................................. 10
  2.2 Tor Browser ....................................... 12
  2.3 Hidden Services .................................. 13
  2.4 Scale of Tor ...................................... 16
    2.4.1 Scale of Hidden Services ................. 17
  2.5 Technical details ................................ 19
    2.5.1 Tor congestion control SENDME’s .......... 20
    2.5.2 .onion name generation. ................ 20

3 Problem Statement .................................. 21

4 Fingerprinting ....................................... 23
  4.1 Overview ......................................... 23
  4.2 History .......................................... 23
  4.3 Adversary Model ................................ 24
  4.4 Theory .......................................... 25
    4.4.1 Support Vector Machine ................... 26
    4.4.2 Difficulties ................................ 27
  4.5 Previous format .................................. 28
  4.6 Countermeasures .................................. 29
    4.6.1 General Idea ................................ 29
    4.6.2 BuFLO ..................................... 29
    4.6.3 Tamaraw ..................................... 30

5 Tominator ............................................ 31
  5.1 Tominator fingerprints ............................ 32

6 Results ............................................. 35
  6.1 Fingerprints ..................................... 35
6.1.1 Gathering Time ........................................ 35
6.1.2 Resources Usage ...................................... 36

7 Roadmap, future improvements 37
7.1 Terminator improvements ................................ 37
  7.1.1 Memory fix ........................................... 37
  7.1.2 Portability ............................................. 37
  7.1.3 Improved Automatisation ............................ 37

8 Conclusions .................................................. 39

Bibliography ................................................... 41
## List of Figures

1.1 Visual representation of what an adversary sees studying the traffic of a Tor-node handling 2 sessions. He is interested in tracking the packet path. But in this case he has 50% chance of guessing which of outgoing packet was decrypted from “12345”.  

2.1 Visual representation of encryption layers of the “onion” by different width and colours.  

2.2 The secret service sets up one (of many) introduction points and sends the *hidden service descriptor* to a HSDir.  

2.3 The user receives the *hidden service descriptor* from the HSDir.  

2.4 The user creates a *Hello*-message, encrypts it with the public key of the service and sends it to an *introduction point*, it contains a one-time password and the *rendezvous point* for the service to establish a connection through. The service may then reply and encrypt the traffic with the one-time password and now they may create a shared secret for future messages to be encrypted by.  

2.5 Picture of direct users every day from June 2013 to April 2014. Exported from the Tor Metrics Portal.  

2.6 Picture of number of relays and bridges from Jan 2013 to April 2014. Exported from the Tor Metrics Portal.  

2.7 Picture of advertised and used bandwidth from Jan 2013 to April 2014. Exported from the Tor Metrics Portal.  

2.8 Picture of used bandwidth by node categories from May to August 2014. Exported from the Tor Metrics Portal.  

4.1 Visual representation of our definition of local adversary. (Figure heavily inspired by Figure 1 in [21])  

4.2 Classifying the space in two sub-spaces with a straight line. To classify dots into their colour-classes.  

4.3 Using the *kernel method* to make a non-linear problem solvable with the SVM. (Figure heavily inspired by Figure 1 in [19])
LIST OF TABLES

2.1 Shows the knowledge of identities by every part of the virtual network. .................................................. 7
2.2 Shows de-anonymisation rate to % of bandwidth controlled, and effect of number of hops. ......................... 7
The year of 2014 has been a very interesting one in the field of computer security and network surveillance. Internet anonymity and privacy have become a hot topic around the world, both in academia and in private affairs. Especially after the revelations of the global surveillance disclosure leaked by Edward Snowden, a former employee of the U.S. National Security Agency (or NSA for short). Edward’s whistleblowing efforts exposed over a million of top-secret documents, many of which explain how the NSA deploy a global scale surveillance of the Internet.

As a repercussion of the leak, anonymity enhancing tools such as Tor formerly known as The Onion Router or TOR, have risen in popularity. By using Tor, the user’s traffic gets encrypted and sent through the Tor-network. The Tor-network is an open-source virtual network on top of the standard network layer, spreading its virtual network nodes across the Internet, far and wide, in a common effort to grant private communication, notably anonymity. Tracking Tor packets across the Internet to their final destination is very hard. First packets may traverse across many different country borders, making it hard for even a national agency to track the full route from start to finish. Secondly during its travel, the packets content will change at every Tor-node as it gets en-/decrypted, making correlation between outgoing and incoming traffic far from trivial. When Tor-nodes handle more than sessions of traffic, it will queue packets and send them in random order, else an adversary could make an correlation knowing the queuing strategy.

Figure 1.1: Visual representation of what an adversary sees studying the traffic of a Tor-node handling 2 sessions. He is interested in tracking the packet path. But in this case he has 50% chance of guessing which of outgoing packet was decrypted from “12345”.

Instead of attacking and monitoring the whole Tor network, by exploiting
known weaknesses or software bugs, this paper focuses on something called *Fingerprinting*. If fingerprinting, a passive attack, was to be successful. Adversaries would be able to unnoticeable figure out which website the *Tor*-user is visiting. This information would be particularly interesting for any general network surveillance system, as otherwise *Tor* would defeat said system. Better understanding how well fingerprinting on *Tor* works, might hint on how to make better future anonymisation systems. Someone who could be interested in becoming said passive adversary under our thread model, would be the secret services of ones state. They could do the fingerprinting either by owning the network infrastructure of said country, or deploy malicious “state approved” software installed on every computer on sale. Using *Tor* might not be a punishable crime, but leaking top-secret files to websites containing controversy and state secret information mostly are. And even in some countries publishing said controversies is considered a crime. A few examples would be reading about the *Tiananmen Square protests of 1989* in China, blasphemy on the Islamic religion in the United Arab Emirates, or general freedom of speech in Iran.

Thus we insist on examining the possibility of developing such a de-anonymising fingerprinting software, and examine the possibilities and ways to make the fingerprinting process harder. Also we theorise on what is needed to make the fingerprinting process impossible for an adversary of this type. Our intention with this paper is to as an independent party, to contribute to the progress of making *Tor* and any other anonymity enhancing software better in the future.

1.1 PERSONAL MOTIVATION AND PURPOSE

Our motivation for doing this thesis sparked out of pure curiosity for the new technology we have at hand. The Internet for many is an endless source of free information, for all people of the world to take part of. Those who are not as lucky have to use means to circumvent censorship, and this could be seen as a good thing out of a pure academic standpoint. As information sharing enables freedom of speech, it can be seen important make it accessible, this is where *Tor* comes into the picture.

*Tor* is a wonderful tool made for people, by people to help put off government censorship and provide freedom of speech for those who might not have the luxury of having it as a right.

By examining *Tor*, we want to help examining its eventual weaknesses, in order to address these. The discovery of a weakness is the first step towards mending them. We want to try making use of our technical knowledge gained through our academic studies to make good.
1.2 PREVIOUS WORK

There are some prior papers on fingerprinting Tor. We found them primarily at “Anonbib” [14], a big list of selected papers on anonymity, driven by the same group that eventually lead to the design and creation of Tor. Most of these papers can of course also be found on library sources such as the “ACM Digital Library” [1].

First off, the one big paper that started all of this: “Tor: The Second-Generation Onion Router” [5]. The original design document for Tor that contains most of the specifics of Tor but also mentions shortcomings such as fingerprinting attacks. Although the design document is over 10 years old now and things have changed a bit, the foundation still stands firm.

Then we have papers on fingerprinting Tor and other anonymity services, such as “Website Fingerprinting in Onion Routing Based Anonymization Networks” [13], which could be considered the founding father of Tor-fingerprinting papers, and one of the most comprehensive studies. In this paper, they created the data-set used by many papers since, getting quite good results on fingerprinting.

From here on, we have a more rapid succession of papers, such as “Peak-a-Boo, I Still See You: Why Efficient Traffic Analysis Countermeasures Fail” [7], which is credited for the BuFLO-countermeasure. “Website Fingerprinting in Onion Routing Based Anonymization Networks” [13], which is a little bit of a continuation trying to evaluate the BuFLO defence of previous paper, and designs a congestion sensitive version with lower overhead.

As an evolution, we have the “Improved Website Fingerprinting on Tor” [22] with its sensational fingerprinting success ratio of up and about 90%. This paper provides the countermeasure of Tamaraw, as well as providing source code and data sets they call traces. These traces is what we would call fingerprints and have been used in previous work [20], although this site with the traces was not released before late May 2014. Same authors also released “Comparing Website Fingerprinting Attacks and Defenses” [21] to compare and evaluate the countermeasures, with the conclusion that theirs was the best one.

These papers showed that it was more than possible to get positive results from fingerprinting attacks on the Tor network, using more advanced methods than previous fingerprinting attacks. More specifically they showed that using a Support Vector Machine (SVM for short), they got far better fingerprinting results than any previous publicly known work. One could assume that big state institutions, such as the NSA, might already reached to the same conclusions before these papers. For them it is an arms-race to find weaknesses before the public, and they would always want to stay at least one step ahead.

These mentioned papers also designed some countermeasures to the fingerprinting problem with various degrees of performance. However, none of these
works released an easy to deploy test suit that would allow anyone to test and obtain own results, or even generate their own data to verify the experiments and results. The tools that they published are not well documented and rather confusing to setup (we tried and failed).

So we decided to develop our own tools for doing the same thing as done in those papers. Foremost because of the scientific method, we want to try to reproduce their results and compare them to ours, and reach conclusions from our work. Hopefully, our tools might be found useful in the future by others.
In this section we describe Tor, previously called The Onion Router with the acronym T0R, but now as a name itself. Tor could be described at various layers of abstraction, as one of the design goals is to make it as transparent and accessible as possible. Tor is the product of the Free Haven Project [15] described originally in “Tor: The Second-Generation Onion Router” [5], published over 10 years ago. Tor is an open source software under the BSD license. Since 2006, it has been developed by the non-profit organisation “The Tor Project, Inc” at the reigns. It is still being very actively developed and also actively researched in a common effort to make it an even better software as every day passes.

In layman’s terms, such as describing Tor to ones non-technical relatives, Tor can be described as a web-browser. You download and install Tor from the Tor project’s web-page1. After installation, you run the browser and it will automatically make you surf the web anonymously through the Tor-network. Quite simply, the program encrypts its users traffic in three different layers of encryption and sends it through the network. The next node in the network will decrypt the packet to get a “next-hop”-address and an encrypted message. It sends it forward to the middle node which does the same and sends it to the exit node which unravels the original packet.

By doing this, Tor tries to protect its users privacy, security and anonymity from all parties who could compromise it:

SERVICE   Tor prevents the service from learning who the user is and the user’s current location. It can be described to works like a proxy server for the users traffic, leaving the service unable to tell who the original user is.

NETWORK    Tor will ensure that the traffic is encrypted, preventing eavesdropping on-route. Also because the traffic is routed to a node of the Tor-network, instead of directly to a service, it will also make it hard to track the packet from start to finish.

TOR        Since Tor is run (mostly) by voluntary efforts, as anyone can become a Tor-node. It thus needs to protect the users from adversaries within the network itself, as any node could be malicious. The trust is distributed in between more parties where none of the nodes by them selves (except you) know the full hop-by-hop route of the traffic.

1 Tor Browser - https://www.torproject.org/download/download-easy.html
Speaking in more technical terms, Tor is a low-latency low-bandwidth encrypted mix-net that consists of voluntary Tor-nodes all around the globe, not “just a web-browser”. At each step of the mix-net, the traffic is encrypted with two separate encryptions. First the Tor-packet is encrypted with three different keys that every node can unlock. And every transaction between nodes are also encrypted with a separate point-to-point using TLS/SSL. Assuming that public-key cryptography itself is strong enough, the encrypted traffic is layered unusable as information to adversaries. And is hard to track as the packets’ changes with each step in the network. There also exists Hidden services which is a part of the Tor-network, which will be more explained in 2.3.

2.1 TOR NODES

The Tor-network consists of voluntary virtual network nodes. Each of which can be classified into different roles, depending on configurations and characteristics. More details on these classifications, see subsection 2.1.1. The uncovering of encryption layers could be thought of as peeling an onion layer by layer to get to the core. This is why we classify Tor as an Onion routing network. This virtual network is developed upon other already existing network architectures, so it does not need any special hardware or software in the infrastructure outside of its own virtual network.

Below we have a representation of Tor, a 4 node linked network. In figure 2.1 we represent our nodes by a capital letter from the alphabet. A, B, C and D. At the leftmost of the figure, we have the user, named A. First A requests for the list of Tor nodes from one of the known directory servers (installed with the software). These contain the information of public Tor-nodes and their IP-addresses, and supplies this information to the user. A can then start a diffie-hellman handshake with B to create a session key for encrypting traffic for A-B. Then B can relay the traffic in between A and C as they want to do an diffie-hellman handshake too.

![Diagram of Tor network](image)

**Figure 2.1:** Visual representation of encryption layers of the “onion” by different width and colours.

A’s identity remains completely hidden to C. With this new connection, A can yet again do it recursively any number of times as A chooses to. But the standard setting is 3, for a good trade-off on security and anonymity versus
latency and bandwidth. Once at the D node, the original HTTP packet gets unwrapped and is forwarded to its final destination, making D look like the origin of the packet.

So the knowledge at each and every node of the virtual network is as follows:

<table>
<thead>
<tr>
<th>Node</th>
<th>Knows Identities of...</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>A, B, C, D, Website</td>
</tr>
<tr>
<td>B</td>
<td>A, B, C</td>
</tr>
<tr>
<td>C</td>
<td>B, C, D</td>
</tr>
<tr>
<td>D</td>
<td>C, D, Website</td>
</tr>
<tr>
<td>Website</td>
<td>D, Website.</td>
</tr>
</tbody>
</table>

Table 2.1: Shows the knowledge of identities by every part of the virtual network.

Assuming we need to know the full path to fully de-anonymise the user, we can calculate the percentage of traffic in the Tor-network that an adversary can spy on in relation to the percentage of owned nodes in the network. The percentage of traffic de-anonymised is calculated as \(P^{n_{\text{hop}}}_{\text{own}}\). \(P_{\text{own}}\) is the fraction of the network the adversary has control over, \(P_{\text{own}} = \frac{\text{ownedNodes}}{\text{totalNetwork}}\), while \(n_{\text{hop}}\) is the number of hops needed to complete the Tor circuit.

With this information we made this pre-calculated table as an illustration of how number of hops plays a role for the level of anonymity.

<table>
<thead>
<tr>
<th>Ownership</th>
<th>1 hop</th>
<th>2 hop</th>
<th>3 hop</th>
<th>4 hop</th>
<th>6 hop</th>
</tr>
</thead>
<tbody>
<tr>
<td>12%</td>
<td>12%</td>
<td>1.4%</td>
<td>0.17%</td>
<td>0.02%</td>
<td>0.0003%</td>
</tr>
<tr>
<td>25%</td>
<td>25%</td>
<td>6.25%</td>
<td>1.56%</td>
<td>0.4%</td>
<td>0.024%</td>
</tr>
<tr>
<td>50%</td>
<td>50%</td>
<td>25%</td>
<td>12.5%</td>
<td>6.25%</td>
<td>1.56%</td>
</tr>
<tr>
<td>75%</td>
<td>75%</td>
<td>56.25%</td>
<td>42.19%</td>
<td>31.65%</td>
<td>17.8%</td>
</tr>
<tr>
<td>99%</td>
<td>99%</td>
<td>98.01%</td>
<td>97%</td>
<td>96.06%</td>
<td>94.15%</td>
</tr>
</tbody>
</table>

Table 2.2: Shows de-anonymisation rate to % of bandwidth controlled, and effect of number of hops.

As easily seen in Table 2.2, to successfully get many full paths the adversary needs to control a high portion of the network. If a single adversary would control 80% of the Tor network, it would only de-anonymise about half of the traffic. Of course, the adversary might be able to get a complete Tor path from a user to the service. As long as the user is using end-to-end encryption like SSH or https the adversary will not be able comprehend the actual encrypted content. The confidentiality and privacy of the data transfer is still safe, but our anonymity has compromised.
2.1.1 Classification of Tor nodes

We would like to mention classifications or roles of our Tor-nodes in the network, and we want to explain these five major classes/roles, before fully explaining them:

- Directory server
- Middle relay
- Exit relay
- Bridges
- Hidden service

To get the Tor network one needs to read the consensus - and this is provided by the network on requests. The consensus is created and is managed by a small Authenticator group preinstalled with the software. This list contains all of the public nodes of the Tor-network, and is called the Tor consensus. A current up-to-date list can be downloaded from the network itself, and can be found on public websites e.g. "www.dan.me.uk, a collection of tools and information" [12] or "https://consensus-health.torproject.org - Tor Project consensus health page" [16].

On Dan’s page the information regarding a node has been through a script for easier human readability, and is presented as following:

<ip>|<name>|<router-port>|<directory-port>|<flags> |<uptime>|<version>|<contactinfo>

And to see a few rows of the populated list, we have chosen to show four of the Swedish non-profit organisation DFRI’s Tor nodes.

171.25.193.131|DFRI2|443|80|EFGHRSUDV|1402146|Tor 0.2.5.2-alpha|DFRI <tor@dfri.se>
171.25.193.20 |DFRI0|443|80|EFGHNRSDV|1690615|Tor 0.2.5.2-alpha|DFRI <tor@dfri.se>
171.25.193.77 |DFRI1|443|80|EFGHNRSDV|1698118|Tor 0.2.5.4-alpha|DFRI <tor@dfri.se>
171.25.193.78 |DFRI4|443|80|EFGHNRSDV|1690493|Tor 0.2.5.4-alpha|DFRI <tor@dfri.se>

A copy of the real consensus file can be found at http://82.94.251.203/tor/status-vote/current/consensus\(^2\) file size of 1379kB last checked to exist 30 Sept 2014. In this, the original data looks more like the text below, and the file is also signed by the Authenticator group to show it’s authenticity.

r DFRI2 bUQxVq1e2GVfHHnQ43CumqQVc c ZS11d5TbWa0AK1ddU14sTj1S51IA 2014-09-30 08:09:23 171.25.193.131 443 80
s Exit Fast Guard NSDir Running Stable V2Dir Valid
v Tor 0.2.5.6-alpha
w Bandwidth=67100
p reject 25,119,135-139,445,563,1214,4661-4666,6346-6429,6699,6881-6999

All nodes have flags, giving them classification on their statuses. Which means that a single node might act as several different classes, except the hidden bridges. As a classification for the hidden bridges really makes no real sense.

\(^2\) The consensus can always found from any directory authority at: http://[directoryauthorityIP]/tor/status-vote/current/consensus
Directory servers exist in two different flavours. The hidden service directory server (HSDir), and the “normal” consensus directory server (V2Dir). The first type contains information about hidden services in form of hidden service descriptors (their public-key, rendezvous points and signature), while the other one contain the public Tor-network and may also have a part of hidden non-public relays called Bridges.

Middle relays count as all the standard relays in the Tor-network. These add to the speed and robustness of the network, while also staying hidden from the final service. All they basically do is to relay traffic that lets Tor-clients connect to the network.

The Exit relays are the actual “connectors” to the normal Internet, the final Tor-destination before the real traffic is unwrapped. They will act as the origin for the traffic to the services, looking indistinguishable from a normal user, like a proxy-server. Exit relays are guaranteed to generate a lot of traffic due to them being a lot less of. Because the exit relays having the original traffic one may also have to give them some additional trust than rest of the Tor nodes, as they may handle un-encrypted and therefore sensitive data. If the original user is using end-to-end encryption, all the exit node really can do is relay the traffic, so it becomes extra important for the user to encrypt their traffic before it is sent through Tor. As otherwise being an exit relay would be very advantageous for an adversary, since there is no guarantee of the traffic being end-to-end encrypted properly, and also to consider in the case where the traffic does not provide forward-secrecy.

Bridges work just like Middle and/or Exit relays, the only real difference is that they are not publicly listed in the list of Tor-nodes. This is to make it harder to block Tor in its entirety, as otherwise you could simply blocking all known IP addresses of nodes listed online in the directory servers. This helps by either adding an extra level of anonymity to whom desire it, both for the node-holder as for users, as well as bridges help fighting censorship.

Hidden services are Tor clients that are also secretly hidden servers, by design only the server itself knows that. The service initialises by generating a hidden service descriptor containing the public key and the introduction points for the hidden service. To ensure identity of the service the descriptor is then signed by its private key. The newly created descriptor is then saved in a hidden directory server in a hashtable, where the hash represents the URL of the website as hash.onion. As seen the address to the hidden service is ending with the pseudo-top-level domain suffix .onion. The hash is 16 character long and is a hash based on the public keys. How the hashes are generated can be read in the more technical part at section 2.5.2. An example of such a hidden site would be http://kpvz7ki2v5agwt35.onion. More information about hidden

Accessible through one of many Tor proxies: http://kpvz7ki2v5agwt35.tor2web.org
services can be read in its own separate section at 2.3

2.1.2 Lookup services

Relays get classified within the network by the network, or flagged as it is called. We list the flags from the specification “Tor Protocol Specification” [4] under section 3.4.1. After the official statement we try to clarify their sometimes too brief descriptions. Some of the additional information is taken from section 3.4.2 from same said specification document.

A router is flagged as “Authority” if the authority generating the network-status document believes it is an authority. This makes us assume that “The Tor Project” chooses trusted points with high bandwidth that becomes authority nodes. All authority servers should have the same network-status document available for download by users, signed by all of the other authorities.

A node would be flagged with “BadExit” if its ISP censors the exit, it’s behind a restrictive proxy, or for some reason akin rendering the exit useless.

Usually because its directory port isn’t working, its bandwidth is always throttled, or for some similar reason that makes it a bad choice as a directory.

In general, a router receives the “Exit” flag if and only if it allows exits to at least two of the ports 80, 443 and 6667 and it should allow exit to one /8 address space. Or more simply, it should be able to use at least two of the common ports for HTTP, HTTPS and IRC. And it needs to have access to at least one of the networks [1 – 255].0.0.0/8.

A node can be marked with “Fast” if it is active, with a bandwidth of top 7 out of 8 (87.5%) of known active nodes. If the minimum bandwidth of the node is at least 100KB/s it will currently be considered as a “Fast” node as well.

4 KB stands for Kilobyte, part from normally standard metrics of kilobits (Kb).
5 This number is adjustable by design.
GUARD  if the router is suitable for use as an entry guard.

Having something called “entry guards” is a design to protect against a few known attacks. The first attack is known as the “predecessor attack” where an attacker would control entry points, and the service. The chance of profiling a user with this attack would be $P^2_{\text{own}}$ as the attacker only needs the “Entry” and the “Exit” node to do timing attacks for profiling its users. As Table 2.1, the attacker could know both information of A and Website at the same time with both the “Entry” and the “Exit” nodes. Instead choosing a “Entry Guard” ones chance to get profiled is the same, but the traffic for a longer while stays hidden. This gives the user a chance of $\frac{\text{totalNetwork}-\text{ownedNodes}}{\text{totalNetwork}}$ to completely dodge the profiling, while the total profiling itself is as large as before.

To gain the “Guard” flag, the nodes Weighted Fractional Uptime must be at least the median for “familiar” active routers. And the bandwidth is at least median, or 250KB/s. A “Familiar” node is a node that have appeared more recently than 1/8 of all active nodes, or it has been around for a few weeks.

HSDIR  if the router is considered a v2 hidden service directory.

A router is a v2 hidden service directory if it stores and serves v2 hidden service descriptors, and that the authority believes that it has been up for at least 25 hours.

NAMED  if the router’s identity-nickname mapping is canonical, and this authority binds names.

As when one installs and sets up a Tor-node there is a option of naming it. Naming the node is easily done with a line in the configuration files in the Tor server settings. Then the Directory authority servers may maintain a file of nickname-to-identity-key mappings. Naming authorities run a script that registers nodes to identity keys if they have been online for at least two weeks, and no other router already has used that nickname for a month. If a node (binding) has not been up for six months, they are removed from the naming file.

RUNNING  if the router is currently usable.

A node gains the “Running” flag if the authority has managed to connect to it successfully within the last 45 minutes.

STABLE  if the router is suitable for long-lived circuits.

A router is considered “Stable” if it is active and either have a Weighted Mean time between failures median, or lower than the median, for known active routers. Or that it corresponds to at least 7 days.
**Unnamed** if another router has bound the name used by this router, and this authority binds names.

A node may gain the flag of “Unnamed” if the name specified by the service already in use by another node. That is, there already exists such a name-to-identity-key mapping.

**Valid** if the router has been ‘validated’.

A node may get the flag of “Valid” if it is running a version of Tor that is not broken. The directory authority may also refuse to give out the “Valid” flag for blacklisted entities due to suspicion.

**V2Dir** if the router implements the v2 directory protocol or higher.

A node may get the “V2Dir” flag if it supports the “v2 directory protocol”. That is, if it has an open directory port, and it is running a version of the directory protocol that supports the functionality needed. This is available in is Tor version 0.1.1.9-alpha or later.

### 2.2 Tor Browser

The Tor Browser (previously known as the Tor Browser Bundle or TBB) is a common effort to make anonymity and privacy available to everyone. It is the flagship “product” of the Tor Project. A pre-configured Mozilla Firefox browser with an automatically starting Tor-service in the background, to make the Tor network accessible as with the press of a button.

The Tor Browser can be run on most major operating systems, including Windows, OS X and Linux. It can be stored and started from a removable and portable media, such as a portable USB thumb drive. This means one does not even need to install anything on the browsing computer, only to make Tor Browser as fully portable as possible.

Tor Project’s main reason to make the product as easy and deployable as possible, is because they want anyone who wants to run Tor, to be able to. By gaining more users, more anonymity is gained. As increased amounts of traffics are mixed, thus it becomes harder to track or be tracked for every user that uses Tor.

In the experimentation and work of this thesis, we have used Tor Browser for all communication with the Tor-network, to get fingerprint data as genuine as possible. We have no reason to expect that using different software and/or methods would not have an impact on the fingerprints and network packets generated.
Being connected to Tor also enables you to reach available hidden services within the network, or .onion-services as they’re known as due to their “fake” URLs. These services are typically special interest websites, exclusive to the Tor-network and not regularly accessible through the standard world wide web. Such a service is easily recognisable by their strange URL, which is 16 random alphanumeric letters followed by .onion.

The unique thing about the connection is that neither one, client nor server, know the physical address of the other. Neither does any node that is relaying the traffic through Tor, most do not even know that they are relaying hidden service traffic. In other words both the server and the client stay anonymous, and no node knows the physical location of the client and the server. Tor can thus also provide privacy and anonymity to the host of the service, as well as the user.

The setup of a hidden service goes as following: First the hidden service constructs a RSA1024 private and public key pair. Then decides and opens a few so called introduction points, which could be seen as open Exit-relays of an open Tor-circuit.

With these things the hidden service creates a hidden service descriptor, which contains the service’s public encryption key and list of introduction points. The hidden service descriptor is then signed by the service’s private key, to prove the ownership of the private key (thus being a legit and not a fake descriptor). The service then sends it to a Hidden service directory (HSDir) server through the Tor-network, which is inserted into the server’s database as shown in figure 2.2, and later obtainable by anyone. Inside the Hidden service directory the descriptor gets saved in its hash-table, where the hash of the descriptor also is the calculated .onion-name of the hidden service. Further how this special hash is calculated is explained in section 2.5.2.

Figure 2.2: The secret service sets up one (of many) introduction points and sends the hidden service descriptor to a HSDir.
From the Tor user’s perspective it starts with them wanting to connect to an .onion site, lets call it abcdefghijklmноп.onion. The user asks a Hidden service directory about a matching hidden service descriptor through a Tor-circuit. If it exists in the database the server will send the descriptor to the user as figure 2.3 shows. Once received the user opens a circuit and sets the last node as a rendezvous point, which will be used later as illustrated in figure 2.3.

Figure 2.3: The user receives the hidden service descriptor from the HSDir.

From the hidden service descriptor the user extracts the public key and list of introduction points, the user chooses one of the points to send a Hello-message to the server. The message gets encrypted with the hidden service’s public key. A Hello-message contains a one-time password and the rendezvous point for sending the Reply-message to the user. The message is sent to an introduction point which relays the message to the hidden service though its connection. The service can now connect to the user through the rendezvous point using the one-time password.

After this “handshake” the host and service are now connected through the Tor-network on a circuit containing 6 intermediate Tor nodes. Neither knowing the physical location (or address) of the other one. The whole progress is illustrated in figure 2.4.

Figure 2.4: The user creates a Hello-message, encrypts it with the public key of the service and sends it to a introduction point, it contains a one-time password and the rendezvous point for the service to establish a connection through. The service may then reply and encrypt the traffic with the one-time password and now they may create a shared secret for future messages to be encrypted by.
Hidden services are typically what we consider a “darknet”. As both the server and the user can stay anonymous to each other, and even to the rest of the network it more than qualifies as one. A side effect of this, the privacy enhancing properties of Tor, is that it may help organised crime. As it has opened up the possibilities for easily accessible black market services such as Silk Road, where people can use Bitcoins to purchase contraband. Even services where users may share and sell underage pornography does exist inside the Tor-network.

For an adversary to match a user with a service through the network, they typically need to first localise the service’s actual location. To do this trivially they need first to own a full path to the rendezvous point as showed on the picture, and then hope they may own the path created from the service to it.

If the adversary gets to control the full path between the user and the hidden service unmasking the service is almost trivial. Controlling nodes up to the rendezvous point is trivial for a smart adversary as you may create your own path, yet the last 3 hops created by the hidden service is up to chance, assuming the service is not smart enough to solely use “safe” paths for itself.

If we assume a powerful adversary does know the physical location of the hidden service, they would still have problem identifying the location of the users, as they’re connecting to it through Tor. Assuming that both the users and the service only use random nodes of the Tor-network, the Table 2.2 shows that even when controlling 75% of the network they can barely expose 20% of the services users. In the real case, probably both the users and services may use some trusted party’s nodes for creating their circuits - as both parties want to minimise the chance of getting caught.

It might be impossible to totally anonymise one-self, yet Tor and its hidden services do provide a far better protection than none at all. Considering that traffic probably bounces through several countries, the adversary would need to not only control high percentages of the national Tor nodes, but also the international Tor nodes.

If not being able to actually own the majority of nodes themselves (due to private and organisational Tor nodes by enthusiasts) the adversary would need to control the networks on which the traffic is transported. Being able to control and spy on the traffic worldwide would be a huge feat, requiring a lot of global collaboration and resources. Most people would deem this as impossible speaking for Tor’s privacy and security, or a risk worth taking.
The statistics below are gathered from the *Tor Metric Portal* [17], these are the official numbers from the “Tor Project” and are freely available to the public. All the figures of statistics are generated on the actual website and exported as PDF to be included in this section. They’re under the creative commons copyright, and free to use.

As can be read by figure 2.5, Tor’s user-base is quite large. The user-base did also recently more than quintupled, some time after *Edward Snowden* revealed about the *NSA* spying program, represented by a big spike in the graph. Since the big spike, it has slowly been on the decline for a bit. Now it seems to have stabilised on slightly over 2.5 million users every day.

![Directly connecting users](https://metrics.torproject.org/)

**Figure 2.5:** Picture of direct users every day from June 2013 to April 2014. Exported from the Tor Metrics Portal.

While the number of users have been fluctuating due to real-world events, the number of relays and bridges have been on an ever so steady incline. In relation to the user-base numbers, it seems to be untouched by the NSA news of last year. As told by reading figure 2.6, we can tell that at the moment of the gathering data. There was slightly over 3000 bridges and 5000 regular Tor-relays operational in April of 2014.

The amount of data transported through the *Tor* network is also on a slight incline with time. As can be read from figure 2.7, the gap in between advertised and actual bandwidth is increasing, meaning that the network should become more ready for an increase in user-base. More importantly that the actual bandwidth of the network has too increased, the current numbers are at 4000 MB/s recorded bandwidth and an advertised bandwidth of 7000 MB/s.
Figure 2.6: Picture of number of relays and bridges from Jan 2013 to April 2014. Exported from the Tor Metrics Portal.

2.4.1 Scale of Hidden Services

Since the actual statistics about hidden services should be hard to reach by design, we can still make an estimation. We have not done our own research for reaching our estimations of the numbers. The number of hidden services and their classifications were obtained from the year old paper of “Content and popularity analysis of Tor hidden services” [2].

In this paper, out of all the 40’000 hidden service descriptors, they found that only a few (3000) were reachable as usable services for them. Most of the hidden service descriptor addresses seem to be part of a botnet on port 55080, as no ports scanned were open, yet left an error message on that specific port.

Out of the normal reachable services, in this case websites, it was found that 44% of these sites were classified as devoted to drugs, adult content or contraband. The other 56% of sites were mostly dedicated to political issues, such as “Anonymity” and “Human Rights”, e.g. reporting and discussing corruption, repression and violations against the concept of freedom-of-speech. Some were to be found described as “Wikileaks”-like sites, discussing and sharing leaked classified state documents and information regarding these.

We figured that we could make a rough estimation on how much traffic the hidden services produces by reading the relay bandwidth statistics as figure 2.8 demonstrates. These numbers will give a very rough estimate of the hidden services traffic, as the optimal goal for Tor would be to hide any statistics possible about the different bandwidth types.

By observing figure 2.8, we read the current bandwidth of the Tor network as 4500 MB/s, this by summarising all the bandwidth of the four classes. We
The only way we would figure to get better values and lower variance would be to monitor all nodes marked as exit nodes. By monitoring these nodes we could mark traffic as Tor and non-Tor-specific one, and note how much outgoing non-Tor-traffic they’re handling to distinguish the bandwidth between guard traffic or actual exit traffic. This would give us better numbers as the classification of traffic would give us better accuracy in the calculation.
2.5 TECHNICAL DETAILS

It is important to note that Tor does not ensure end-to-end encryption, which is commonly seen as the ultimate tool in privacy to keep information confidential, and can also provide integrity and authenticity of the information.

Much of the following information has been taken from the “Tor Protocol Specification” which was fetched and is available on github [4]. As Tor is an actively ongoing project these specifications can change at any time in the future.

Each connection between each node of the Tor-network, including the client to the entry point is encrypted with TLS/SSLv3 for the link authentication and encryption. The lowest encryption suite that must supported is the SSL_DHE_RSA_WITH_3DES_EDE_CBC_SHA cipher suite, but most preferably the connection should have support for TLS_DHE_RSA_WITH_AES_128_CBC_SHA. The traffic mimics popular web-browser TLS/SSLv3 traffic, which usually support a longer list of ciphersuites that might support longer keys, such as TLS1_ECDHE_ECDSA_WITH_AES_256_CBC_SHA.

The basic mean of traffic in between nodes is sending information in fixed-width blocks, called “cells”. Blocks are 512 byte long with its own header and payload area. These blocks are encrypted with a stream cipher, which is the 128-bit AES in counter mode.

As mentioned before, the traffic through the Tor-network gets encrypted by the TLS protocol with a 128-bit AES stream cipher. The encryption happens in two different layers. The first layer is point-to-point encryption in between each node of the routing, that is A-B, B-C and C-D. The second layer which is

Figure 2.8: Picture of used bandwidth by node categories from May to August 2014. Exported from the Tor Metrics Portal.
the pure “onion” is encrypted as A-B, A-C, and A-D, as represented by Figure 2.1. Thanks to the multiple layers of encryption, on both the “onion” and node-to-node encryption, this ensures perfect forward secrecy for the Tor-traffic [5].

2.5.1 Tor congestion control SENDME’s

Tor also uses congestion control, since individual bandwidths are a limited resource, we will need to care about congestion, both accidental and intentional congestion. This is why Tor by design has included both point-to-point-level and stream-level throttling.

There are two types of throttling, point-to-point throttling and stream-level throttling. That is for each layer. So the end-to-end throttling (that is OP-OR ⇔ A-B, B-C and C-D) works that the windows start with an initial value. Then for every packet the window is decreased by one. Every 100 data cells OR receives, OR sends OP a relay sendme cell with the streamID of 0. When received OP increases it’s windows by 100. If the window reaches 0, the OR stops reading all connections on the circuit.

The stream-level that is A-D congestion control is similar, where the initial window is 500, and the size of increase and sending of relay sendme’s with streamID equals 0, is 50 instead of 100.

2.5.2 .onion name generation.

When you decide to run a hidden service Tor generates an RSA-1024 keypair. The hash in hash.onion is calculated as “the first half of the Base32 representation of the SHA1 hash of the DER-encoded ASN.1 public key”. In programming terms, it would be described as this pseudo-code which we deem much easier to read.

\[
(K_{\text{priv}}, K_{\text{pub}}) \leq \text{MakeKeypair().ASN1()}
\]
\[
\text{StorePrivate}(K_{\text{priv}})
\]
\[
\text{DER} \leq \text{ExportDer}(K_{\text{pub}})
\]
\[
\text{Hash160} \leq \text{SHA1}(K_{\text{pub}})
\]
\[
\text{Hash32} \leq \text{Base32}(\text{Hash160})
\]
\[
\text{onionHash} \leq \text{Hash32.getFromTo}(0,15)
\]
\[
\text{output(onionHash)}
\]

SHA1 creates a 160bit digest of the public key, and since a character in BASE32 consists of 5 bit of information, means that the BASE32 representation of the SHA1 digest is 32 characters long. This the first half of this is then what we prepend to .onion to get the URL.
PROBLEM STATEMENT

With Tor anyone (e.g. journalist, rights activist, curious civilian) could access blocked websites within a restricted network. An agent could leak information on findings within state secret organisations, or a journalist could write about the current situation in the country. With the cover of anonymity provided by Tor the restricting party would have a hard time pinpointing the specific person.

Tor is a great tool to give people the power of anonymity against surveillance on the web, but it is not without flaws. As previous literature has been pointing out, there are several Tor vulnerabilities that must be explored. Especially one that requires less overall control of the complete network: fingerprinting attacks. If they are very successful, they could be used to de-anonymise these users, leading, for example, to certain conviction and imprisonment.

Website fingerprinting attacks are classified as side-channel attacks. That is, in a sense, reading side channel information that is created or exists in an encrypted channel. We can see encryption as a function where the domain $X = \{\ldots\}^n$ is a bijection into the domain $Y = \{\ldots\}^n$ with some function $f_{\text{key}}(X) \rightarrow Y$, with the unique inverse function $f'_{\text{key}}(Y) \rightarrow X$. As the function works uniquely for any key, we can not find the reversing function of elements of the same length. Not knowing the content we know the elements length. This increases the uncertainty of an adversary, as the information length can not be pinpointed. With a larger padding, we achieve a greater effect of uncertainty, but at the cost of overhead of the encryption.

Even though it seems a hard task, people have figured out that browsing a website through Tor yields unique enough fingerprints to distinguish large sets of websites. This is because websites have often different traffic patterns, due to how the data is retrieved and the size of each and every chunk. The response time for different servers are different, and so is the size and ordering of packets. Even for two identically large websites, in comparison of data being transferred. We can find unique exchange orders in which how the data is retrieved and asked for.

A fingerprinting attack can be done completely passively and on a very large scale. The adversary being interested in a Tor user’s browsing habits can passively listen (wiretap) to the information exchange. As the attacker does not need to inject anything or corrupt the traffic, this leaves the Tor user unaware.
of the attack ever happening. As the Tor can’t detect the fingerprinting attack being run or not, we need to develop countermeasures that are constantly running.

The literature has provided a number of works but with few open tools for anyone to use. Tools available are very hard to grasp, and the fingerprints and data-sets used in these works have done little to convince the reader that they are not just generated in a lab setting. This work can then be used as a stepping stone for future research on fingerprinting of Tor, to explore the problem of fingerprinting, thus also getting a great tool for examining countermeasures of the attack.

That’s where we come in. We want to try to use the same pieces of theory and recreate their results with our own analysis. Instead of using the same data set, we want to make our own fingerprinting suite, Torminator (more in chapter 5). While on it also improving on the longevity of the design of fingerprints brought by earlier work by including extensive side-channel information. Including also date and site in the actual fingerprint, while being backwards compatible with some minor changes.

We also want to touch on the subject of countermeasures against fingerprinting attacks. And the actual matching with fingerprints done with machine learning, for example with a support vector machine (SVM) as previously done by other researches.

With Torminator, we also want to create a dataset that can be used by future research. Instead of forcing them to actually use our program and make their own fingerprints, we can support them with our ones on request. We want to apply usage of our expertise in the area opening the doors for future works.
A fingerprint in computing could be considered just like a human fingerprint, a certain set of features that uniquely identifies its owner. One type of fingerprint many have encountered is hash algorithms that map an arbitrary large data item into a much shorter digested bit string. This is often used in corruption checking of files (does the hash match?), or integrity of data using cryptographic signatures.

In our case a website fingerprinting is when we read features of a website to be able to uniquely identify it, this without actively reading the data itself. This is because the encrypted traffic still generates readable side-channel data. Side-channel data is in our case are the characteristics of the traffic itself, and not the contents, characteristics such as: latency, size and bandwidth.

Some early work theorised about the limitations of end-to-end encryption: leaked features of the traffic such as response time and size of the transmitted data could be enough to compromise the traffic [9]. It was theorised that reading the side-channel information could be enough to gain information about the encrypted content of end-to-end encryption systems. As an example, web browsing through an SSH-tunnel or a proxy-server does not guarantee the confidentiality of the information when one could correlate traffic signatures to certain websites.

It’s not that simple when considering Tor. The latency is randomised due to the random path the traffic has to take; the same is true for the the bandwidth. Because these two pieces of information are so varied, fingerprinting becomes a far less trivial task on Tor-traffic. In recent years, researchers claim that they have conquered this difficult task and are able to deploy fingerprinting attacks on the Tor-network. And with every year, the methods are getting ever more advanced with higher fingerprinting precision. Quite recently, the paper “Improved Website Fingerprinting on Tor” [22] has been noted for getting a huge recall rate on their fingerprinting by implementing a Support Vector Machine.
(SVM) to fingerprint the Tor-network. With the SVM getting incredible accuracy of about 90%, exceeding all previous known methods. This sensational paper is the single most influential one for this thesis.

### 4.3 Adversary Model

The adversary model we have chosen for our paper is a passive local adversary that has access to the encrypted traffic in between the Tor-client and the first hop (entry guard) of the Tor-network. This adversary could be a dormant program running under the operating system, e.g., like virus or spyware\(^1\). Or, the adversary could be anyone having direct access (tapping on) the network traffic. Like the Tor-user’s ISP or the national security agency of ones country, or a user on the local network. One could also consider the ISP of the entry node for the Tor user. Or even the node itself could be considered an adversary. Simply put, anyone who can access a copy of the encrypted traffic travelling to and from the Tor-client to the first node of the network, as illustrated by figure 4.1.

![Figure 4.1: Visual representation of our definition of local adversary. (Figure heavily inspired by Figure 1 in [21])](image)

This particular adversary model has been chosen as it could be easily deployed by anyone, and it is the only point where we can trivially know exactly who the Tor-user is. If the adversary would be deeper into the network, such as in between B–C or C–D as on figure 4.1, the adversary can not possibly know who the client is, but could be used to gather statistics of the network. And if the adversary would be the exit node (D), or between it and the service, the original client would be safe as long as the user is using end-to-end encryption (and used correctly), rendering it no better than the C-spot.

As the traffic enters the Tor-network, the user becomes anonymous to the

\(^1\) We assume that they only have access to read the encrypted traffic - but in the typical case they can do better than that.
network itself and the service connected to as we showed in Table 2.1. The
original packets that are unwrapped at the exit node do not contain the normal
TCP/IP information of the original host i.e. IP address. The information of
the origin of the original network packet is not attached through the Tor-circuit
and lost to passive adversaries lurking by the exit node. This is why we render
our adversary model as the best spot for a passive adversary to attack from, as
other parts of the network would impossibly identify and differentiate traffic
for a Tor-user.

4.4 Theory

The general theory of fingerprinting still stands, and due to the nature of Tor
being a low-latency low-bandwidth service, there will be side-channel information
up for the taking, as all browsing of all websites won’t create exactly the
same traffic and patterns. Although the fingerprints in our current model are
limited to webpages, not for complete websites. That is, the content of a page
generates the fingerprint, not much as side effects of the servers as done in ear-
lier fingerprinting works on other services [8]. The only side-channel data we
can reliably use is the packet sizes and directions, but we deem that it could be
interesting to save date and relative timings as well. In previous works it was
only considered saving the packet sizes and directions, this can be very incon-
stant over a full site, but somewhat constant for a (non-altered)page. Here We
only examine single web pages, and those are the standard index-pages from
a list of most visited websites on the web.

There is a limitation on fingerprinting servers through Tor, and previous
work has not worked with this. This is because of the high variance on band-
width and delay, even when using a single route. This essentially removes
the possibility of collecting the side-channel information towards server finger-
printing.

One could possibly use other characteristics of the data hoping to recognise
what type of data being sent to the client. For example, in a setting where a
user streams video or audio. As these streaming services send out the data in
short bursts over a long time while the user is watching the video, giving a
certain pattern for the data. As otherwise it would be very memory expensive
for long videos to be saved on the users computer, either in memory, or hard-
drive. These characteristics are common in video sites, such as Youtube and
with this information we can easily spot a user watching a video by reading
network pattern of the user traffic.

To recognise such patterns we may use different machine learning algo-
rithms. As explained in 4.2, currently the most successful one for fingerprint-


ing sites through Tor is the Support Vector Machine.

4.4.1 Support Vector Machine

A Support Vector Machine (SVM) is a supervised learning model, which uses learning algorithms to analyse and recognise patterns. This is typically used either for classification or regression analysis. A supervised learning model means that it needs to be "taught" - this by giving it classified data called the training set. Once taught the SVM may use what it learned to predict classification for unclassified data from the testing set or by using regression analysis predict future data.

SVMs have and can be used for many different applications, wherever prediction and classification is needed. It has been used in Protein Structure Prediction [10], and detecting steganography in images [11] to name a very few.

Classifying could be simply thought of as having a space where we want to classify points of data into one of two classes. For example dots on a plane and with different colours. A classification in machine learning would be when we separate them with a straight line, leaving them in two different spaces with each class on either side of the line as illustrated by figure 4.2.

![Figure 4.2: Classifying the space in two sub-spaces with a straight line. To classify dots into their colour-classes.](image)

On harder problems the classification problem no longer is solvable by linear means, we may apply the kernel method to transform the data into a higher dimensional feature space. In which the problem can be solved by separating the data with a hyperplane, instead of a line, as illustrated by figure 4.3. Thus using the kernel method (sometimes called kernel trick) we can also use the SVM to solve non-linear classification problems.

The SVM may also be extended into a multi-class classifier by implementing decision-trees, extending the problem into multiple binary classification problems [6]. By doing this, we would theoretically be able to make the SVM take in a set of data traffic side-channel data, and make it classify the traffic as a certain trained website.
4.4.2 | Difficulties

Below we will mention a few difficulties in gathering fingerprints for sites.

**Freshness**

A website might change over time as most know. The contents on the page might not be the same the next time it is visited. However, one thing is working for us, and that is Tor Browser’s cookie-less nature forces everyone’s page to standard instead of personalised on first entry. This is so because cookies are the identifiers and the way to hold and remember sessions in the HTTP-protocol. And since saving an identifier typically is not desired and works against the anonymity for user’s they’re quite absent from the Tor Browser. Otherwise a very simple attack against Tor Browser would be malware that find and read its cookies.

Since the browser is cookie-less at the start of a browsing experience by default, the browser and users will get directed to the startpage of the requested site when going to a typical URL. Meaning that sites whose content is more varied once logged in, will stay typically be more static, and in some cases only showing a login page.

It is then a completely different problem to evaluate the freshness of a page. Some pages have huge fluctuations of content all through the day like a livestream of tweets on a popular “hashtag”. While some other webpages change very seldom and could almost be seen as eternal, like blog-posts, wiki pages, and login pages for popular sites.

**Content generation**

Many sites use content generation, which affects the freshness of a site. This could be for example localisation of a site (connecting to www.google.com gets
redirected to www.google.se in Sweden). Some sites are run by handling random-content like http://randomtextgenerator.com/ or https://en.wikipedia.org/wiki/Special:Random (two different connections at the same time receive different contents).

**Uptime**

One error that does occur while doing fingerprinting through the Tor network, is that the connection fails or is too slow to give the site. It might also be that the site searched for is down, either due to service or malicious attacks such as a DDoS. Simply put, if we don’t have a connectivity to a site, we can’t possibly fingerprint it.

### 4.5 Previous Format

The format of fingerprints used in previous works is the so called “Traces for open world (WPES 2013)”. The fingerprinted website is identified by a number, such as 1, and the fingerprints are stored in files on the format number_ID.txt where ID typically is a unique identifier, in this case fingerprint number ranging from 1 to n where n is the number of fingerprints.

Example the fingerprint from the set opencelltraces0-0 that is called 1_1.txt contains the following lines.

```
600
600
-600
600
-600
600
-1800
-1200
600
600
```

These fingerprints are then used as input to the machine learning part of the fingerprinting. Either as training-set or the testing-set. Sadly this format lacks much on the human-readability, although can work very well in a lab setting. And it does neither tell us from what site (more than “1”) fingerprint was created from, in case we want to create our own and compare.
4.6 COUNTERMEASURES

4.6.1 General Idea

The general idea to thwart fingerprinting is to “simply” remove or alter what makes fingerprinting possible, the side-channel data. The removal of side channel data is easy, but comes at a very high cost. For a low-latency low-bandwidth network such as Tor, it is not desired by the users lose either of the low-latency or low-bandwidth arguments. Both of which are very important for the functionality of browsing the web. One needs to somehow compromise and find nice solution that gives as little overhead in them as possible.

A very simple solution would be to make the bandwidth for traffic constant. Even when no real data is sent we just generate some random dummy data to be sent. This would require lots of extra “meaningless” traffic, but the adversaries could not read the speed nor size of the data sent, making it really hard to fingerprint (as the side-channel data is altered).

4.6.2 BuFLO

BuFLO (Buffered Fixed-Length Obfuscation) comes from “Peek-a-Boo, I Still See You: Why Efficient Traffic Analysis Countermeasures Fail” [7], and is described as ”idealised best case” defence against fingerprinting attacks. It is NOT designed with Tor in mind, as the paper is about fingerprinting in general, not just Tor. As it is designed for the general case, the part where we send data in fixed-lenth packets don’t make sence, as Tor does this by design.

This is just a more detailed specification of how it would be implemented. It is therefore an inefficient countermeasure when it comes to a low cost overhead and could be considered strictly made for comparing defences against. The specification in the literature, gives BuFLO three settings integers: \(d\), \(\rho\) and \(\tau\).

\(d\) - \(d\).

The size of the fixed-length packets. The larger this value is, the more overhead and protection is given.

\(\rho\) - \(\rho\).

The frequency (in milliseconds) in which the packets are being sent. This has a large impact on the bandwidth and responsiveness.

\(\tau\) - \(\tau\).

The minimum amount of time (in milliseconds) for which we must send packets.

When we establish a connection we will send packets as soon as we have filled out buffer to create \(d\)-sized packets. Whenever or not we have a packet
ready we send one at the rate of $\rho$ (milliseconds in which packets are sent). When we don’t have enough in our buffer, we send a fake packet with random data in it. This creates a constant stream, during our communication of at least $d \cdot \rho$. The communication lasts for at least $\tau$, so if we’re finished early we will send $d$-sized packets every $\rho$ until we meet $\tau$. And if we have surpassed $\tau$, we can simply terminate where right away. In the paper it is shown that this countermeasure works against fingerprint, but it will give a lot of bandwidth and delay and protection depending on the setting of the integers.

4.6.3 Tamaraw

The Tamaraw$^2$ defence is designed and presented in the “Comparing Website Fingerprinting Attacks and Defenses” [21] paper. A much more efficient countermeasure than BuFL0 against fingerprinting attacks. It is also stated in the paper that it gives better protection than all previous defences.

In this defence they expanded the rate of $\rho$ to both in- and out-going traffic, $\rho_{\text{out}}$ and $\rho_{\text{in}}$. And a padding parameter of $L$. This $L$ is a non-trivial choice, and if it is large enough then it is very likely that there is another web page that is mapped to this same multiple of $L$. And in their experiments it is shown that this design provides a much better protection-rate to any loss of bandwidth.

---

$^2$ A tamaraw is a small cousin of the buffalo, endemic to the Philippine islands.
We programmed our own Tor packet capturing software in the Java programming language, and we call it Torminator. The name as many might recognise is a mash-up of the evil robots from The Terminator franchise and Tor. Torminator is quite a simple program that uses the jNetCap, a Java library, which uses the native C-library libpcap [18].

In Torminator we capture all network traffic, and try to sieve out which packets are Tor-traffic, and after identification to read and extract side-channel data from them. The data we gather is: Web page URL, timing (also ordering and keeping date), packet size and direction of every packet we deem related to the Tor-network.

We also made a few shell-scripts that Torminator will execute from within, to do things that Java have a harder time to harder to do. Things like simulating a user, using the official Tor Browser, this to give us as genuine usage traffic as possible. In the script we start the Tor Browser, type the web page name into the address bar, and hit the Enter key to browse to the website. After a set amount of time, the script closes the tab and exists the browser.

These actions are then combined with our program, which reads from a list of popular websites and sends orders via the script to visit them. By doing this we have made an automatic fingerprinting gathering software that tries to be relatively true to the nature of a human browsing the web.

A question that has been asked is why we didn’t make the process easier, scripting to use simpler programs such as wget or curl to fetch web pages through the Tor network. The answer is that we don’t want to risk the network traffic being different due to us using other programs than the official ones provided by the Tor Project. Even though me make a few configurations that are non standard to acquire more speed and ease, we are sure that the traffic stays authentic.

To make the fingerprint gathering program a little bit easier and faster to execute, we configured our Tor-client to only use the Swedish non-profit organisation DFRI’s entry nodes. This because we could then ensure that the first node chosen has quite low latency and quite high bandwidth to be used by us. It also was very simple to code, as we could fetch their servers’ public IP addresses once, and save them in code. Making it easy to identify Tor traffic.

---

1 DFRI - Freningen fr Digitala Fri- och Rättigheter - https://www.dfri.se/.
We mostly used these to assure ourselves some speed increase as running long simulations with slow speeds would give us worse fingerprints as we only want to run a single fingerprint for a maximum of 45 seconds\(^2\) - as we failed to set-up in such a way that the browser quits as soon as it loads to save time.

### 5.1 Torminator Fingerprints

Most of the previous papers have been vague on how they formed their fingerprints for their tests. But searching through the web, we have found the fingerprint files from “Touching from a Distance: Website Fingerprinting Attacks and Defenses” on T. Wang’s webpage [20], one who has been part of a few of mentioned papers before such as the “Improved Website Fingerprinting on Tor” [22].

These fingerprints as previously mentioned are named SITE\_ID, this probably to keep the information of the fingerprints simple. SITE is just a number for identification and ID is for identifying separate files. A quick analysis gives us that they’re formed as following,

Row 1: NUM
...
Row N: NUM

Where NUM is the packet size and direction of the traffic, direction being determined as the positive or negative value of the packet size. Most of the previous papers seem to use these fingerprints, which are a little hard to read and to be found.

We design our own new fingerprint format, and in our design we put in a bit more information. Our specification is as follows:

Row 1: Website: \$WEBSITE
Row 2: NUM DATE:milis
...
Row N+1: NUM DATE:milis

Just to give a living example, here are the ten first rows of one of the fingerprint files. In this case fingerprinting the browsing of the site http://en.wikipedia.org during the fifth of May 2014.

Website: http://en.wikipedia.org
-585, 05-05-2014\_17:29:04:374
20, 05-05-2014\_17:29:04:376

\(^2\) This value is by all means configurable.
The way these fingerprints are made has been chosen to closely remind of the fingerprints that are provided in the previous papers, but with some improvements. Our data tells which website it is within the data of the fingerprint, which is very usable when we’re making a training set. And instead of just calling them by numbers we have saved the actual URL’s for the fingerprint. In precious work they have only been numerated by numbers, which is not as clear for an external party.

The other one being time and date, followed by millisecond timing. Keeping date information is important due to freshness issues explained in 4.4.2. To get greater fingerprinting results, we want fingerprints in the database to be as close in freshness as possible to the attacked packets. The timing information could be cleverly used as the timing can be used for non-realtime fingerprinting systems, as it is not clear that the inter-packet timing is completely useless. That is, we can gather fingerprints, and a user’s Tor traffic data in different systems at the same time, and later try calculating which site the user was browsing to. And in worst case, we may wait for our fingerprinting system to become more advanced, and in retrospect defeat the anonymity of a user.
To make a genuine fingerprint, one needs to make actual traffic through the actual Tor-network. Tor has itself a limited bandwidth, which could be considered as one of the big out-of-our-control bottlenecks of making fingerprint learning-set data on a massive scale. For those having greater interest and resources in gathering fingerprints, they could of course just make their own nodes that they supply with bandwidth and set-up all their traffic to go through these nodes. This would make it possible for people with high resources to make large amounts of fingerprints. The bottleneck would then be the bandwidth of the site which is the fingerprinted. In which case we can just apply parallelism and connect to a multitude of other sites we want to fingerprint at the same time.

To gain good results in a machine learning setting we typically need to gather a multitude of fingerprints to get good recall values. In our case getting one fingerprint for one site through the Tor network takes about a minute. We are limited to the main page of a site and we’re not fingerprinting any sub-pages of the website. This is because we deem fingerprinting through Tor as limited to know which server we’re connecting to, as we could figure out in a traditional setting. As for a single adversary with very limited resources, such as our adversary, the gathering time of enough fingerprints is large and could take days to complete for a large set of sites. This because we would consider that it is needed to fingerprint a lot of different web-pages, and many times each to gather enough data to teach a machine well. But in the case we would only want to know if the user is surfing a single page or not, it would be more than feasible in a live system.

Let’s examine our 250,60-set, which is 250 different web pages and 60 fingerprints each, that’s 15000 fingerprints. Reading the dating on the first recorded fingerprint “21-04-2014@11:29:54:187” and the last recorded one “30-04-2014@18:21:36:321” reveals that the gathering took roughly 223 hours to complete.

Our single largest and longest run was the 250,60 run. But we have other
notable ones such as the 10,1000- and 1000,10-sets. That is fingerprinting the
top 10 sites 1000 times, and the top 1000 sites 10 times. These 10000 fingerprint
sized sets took roughly a week of simulations each to complete.

We did consider using the Weka machine learning software for our SVM needs,
but to make it work we need to convert our fingerprint files into compatible.
.arff files. We made a simple script to convert the information in the terminal,
by using a simple Java program we made. This script can convert about 5
to 10 lines (or packet representations) every second using a bash script that
extracts the data from the raw data files and translates them into the required
.arff-structure. The code for the Java-program makeArff.jar is attached in the
appendix B.

6.1.2 Resources Usage

Disk usage

Each packet in the data stream has been represented by a string of its size,
direction, and detection date. A collection of these packet’s meta-data for a
single web page download is what a fingerprint consists of. The typical string
for a packet has a size of 28 to 31 characters. Each fingerprint ranges from
hundreds to several thousands of packets. We can make a rough calculation of
our average fingerprint size (including failed packets) from 10’000 fingerprints
from our 1000_10-data as \( \frac{\text{TotalSizeMB}}{\#\text{NumberOfFiles}} = \frac{700\text{MB}}{10000} = 70\text{KB} \). Since we don’t use
any special characters the encoding of the files as plain text will be 1byte per
letter in our UTF-8 files. We can easily calculate that each file on average has
70000 characters, thus an average of \( \frac{70000}{29.5} = 2373 \) lines for each fingerprint file.

Due to the large amount of redundancy in the information in the files, one
can typically compress this information with a compression algorithm. In our
case we compressed to 7z\(^1\), a free and open source archive file format that uses
the Lempel-Ziv-Markov chain algorithm LZMA. This gave us a compression rate
of around 5% of the files original disk usage, this saves a lot of disk space, but
comes at a cost of reading speed for the machine. And as all our fingerprints
while compressed have a file-size of 179MB, compressing a grand total of 65792
files or 3.4GB of fingerprinting data.

---

\(^1\) 7zip.org Official project web page - [http://www.7-zip.org/](http://www.7-zip.org/)
ROADMAP, FUTURE IMPROVEMENTS

7.1 | TORMINATOR IMPROVEMENTS

7.1.1 | Memory fix

One problem that has bothered us is that the program does crash if it runs for too long (periods of weeks). The error seems to have been pinpointed to that we’re out of memory on the VM. Leading to a stop of network capturing, while still Torminator is running, this leading to empty fingerprint files. A quick analysis gives us the hypotheses that there might either be something wrong with jNetPcap, or with with the inner workings of Torminator. A quick check on jNetPcap makes us find a website telling about a fix “Out of Memory” errors (such as we’ve gotten) that might be relevant at http://jnetpcap.com/?q=node/1193 (August 2014). One could also rewrite the whole program in a more efficient language like C or C++, then handing the memory manually. This leading to if coded correctly, manually leading to that there won’t be a out of memory error.

7.1.2 | Portability

Torminator is dependant on libpcap being installed on the system. So even if we use Java as our main language to make it portable, we still lose the portability as the portable .jar file we can export won’t run on systems without it. Moreover, we use Bash-scripting sessions in our program to handle functionality as steering the Tor Browser. Leading to even more portability disfunction. Inside these we also use programs native to Linux, neither being installed on a standard distribution. A huge cleanup and portability effort could be made to make Torminator run on all three major operating systems, improving the portability of the program.

7.1.3 | Improved Automatisation

We could have made the Torminator work for any (public) Tor traffic by fetching the Tor consensus at run-time. Meaning the list of all public Tor nodes and their IP addresses. This could be implemented either by reading the in-
formation Tor itself uses, or get it one-selves from one of the Directory servers. This would have made our program independant of DFRI, and also would have provided better portability.

A thing that would increase our speed of gathering fingerprints, would be if the web browser would close once the loading of a page is done automatically. As we mentioned, we close after a set amount of time, no matter if the page has fully loaded or not. And some sites are very quick to load, we still have to reach our static threshold, which leads to a longer wait than necessary. If we or the browser could detect and close once all elements are loaded, this could give us far more efficient way of fingerprinting fast loading sites.

Yet another thing that would give us more fingerprints faster, is to introduce some level of parallelism. In our case we could cut the gathering times from weeks to days by having a multitude of computers doing the fingerprinting attack. Although too much traffic would put strain on the Tor-network. The problem we have here is we would need more machines (or virtual machines) as one can not run many instances of Tor browser. It has not been explored by us, but maybe one would load more sites at the same time and distinguish the traffic for each browser-tab?
At the end of this thesis work we have come to the conclusion that it is quite an easy task to create fingerprints of Tor-traffic. Capturing and filtering the network packets are easy tasks, but there seem to be no papers before that have tried to standardise the fingerprint layout for future research. That is, in a sane and well-motivated manner that have caught on.

We have created Torminator, a Tor fingerprinting suite written in Java, to make it easy for anyone interested to start creating Tor fingerprints. It is still in it’s early stages of development, but it lays down a good ground for future research in Tor fingerprinting. Torminator has been used to create Tor fingerprints, which provide the essential information needed to preform de-anonymising attacks on users of Tor.

With the information in these fingerprints, with some minor tweaks, one should at least be able to preform said things in previous work. But the fingerprints designed for Torminator contain more information, which might be usable for even future research. This further information is interpacket timing, dating and website URL which is not shown in previous works’ fingerprints.

We have also made a large amount of fingerprints with Torminator, as many as 65792 fingerprints, spanning over the thousand most popular webpages, with much focus on many fingerprints on the ten most popular ones. This data-set could be provided for future fingerprinting research as it follows the newer more sane fingerprint standard. We would like to see fingerprinting work in the future utilising our fingerprints, confirming previous works results of that the fingerprinting attack is viable and highly accurate.
BIBLIOGRAPHY


GLOSSARY

**NSA** - National Security Agency
American intelligence agency specialised in global monitoring.

**SVM** - Support vector machine
A term in machine learning, a group of models and learning algorithms to analyse and recognise patterns in data, used for classification and regression analysis.

**TOR** - Formerly, TOR - The Onion Router
Is a virtual network, and software used to browse the network with a higher grade of security and anonymity than by standard means. Tor is simply the name of the service.

**.7Z** - A compressed archive file format
Our choice of archive format for archiving and compressing our data.

**API** - Application programming interface
A specification of how to interact with a piece of software or hardware using a programming language.

**ISP** - Internet service provider
The entity that owns and provides the costumer with his/her Internet connection.

**DFRI** - Freningen fr Digitala Fri- och Rttigheter
DFRI is a Swedish nonprofit organisation working for digital rights.

**RSA** - the letters initials of creators surnames.
Is a very common public-key cryptosystem, and is based on the practical difficulty of factoring the product of two large prime numbers. Publicly described in 1977 by Ron Rivest, Adi Shamir, and Leonard Adleman.

**VM** - Virtual Machine
Is an emulation of a particular computer system. It can be used to run multiple instances of operating systems on a computer, or running programs in portable programming languages such as Java.
**DNS** - Domain Name System
Works like a phonebook for computers, binding “dynamic names” to static IP address numbers. So if servers moves or changes address it will still be reachable with the same DNS.

**SUNET** - Swedish University Computer Network
An internet service provider and organisation, responsible for the Swedish Universities network.

**KTH** - Kungliga Tekniska Hgskolan
Swedish for The Royal Institute of Technology, located in Stockholm, Sweden.

**IP** - Internet Protocol
Usually combined (though often left out) with “Address” for Internet Protocol Addresses. The dominating addressing system for computers, IP version 4 and upcoming IP version 6.

**SSL/TLS** - Transport Layer Security and Secure Sockets Layer
The cryptographic protocols designed to provide communication security over the Internet. SSL is the predecessor for the improved TLS protocol.

**AES** - Advanced Encryption Standard
The encryption standard for electronic data, established 2001 by the United States’ National Institute of Standards and Technology.

**HTTP** - Hypertext Transfer Protocol

**HTTPS** - Hypertext Transfer Protocol Secure
Technically not a protocol at all, but the layering of SSL/TLS and HTTP protocols.
Script to convert all regular fingerprints *txt into *.arff files:

```bash
#!/bin/bash
OUTPUTFOLDER="./"
for f in *.txt;
do
   java -jar makeArff.jar $f $OUTPUTFOLDER
   rm $f
done
```

Source for the Java-program to convert our fingerprints into Weka specific files:

```java
import java.io.File;
import java.io.FileNotFoundException;
import java.io.PrintWriter;
import java.util.Scanner;

/**
 * This package was made to make my datafiles
 * into arff files that can be used in Weka.
 * I have chosen my ARFF files to be represented by
 * ID,DIRECTION,LENGTH,DELAY
 * @ID − Number in sequence of packets
 * @DIRECTION − packet direction. IN or OUT
 * @LENGTH − the length of the packet.
 * @DELAY − the total delay since last sent/received.
 * @relation tor−fingerprint
 * @attribute ID INTEGER
 * @attribute DIRECTION {IN,OUT}
 * @attribute LENGTH INTEGER
 * @attribute DELAY INTEGER
 */
public class ArffMaker {
   private static PrintWriter writer;
```
private static Scanner reader;
private static int globalCounter = 1;
private static int lastDelay = -1;

private static String outputDestination = "/";
private static String website = "";

public static void main(String[] args) {
    if (args != null && args.length != 2) {
        return;
    }

    File inFile = new File(args[0]);
    outputDestination = args[1];
    if (!inFile.exists()){
        System.err.println("First parameter of in file, does not exist");
        return;
    }

    try {
        reader = new Scanner(inFile);
    } catch (FileNotFoundException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }

    try {
        String outFile = args[0];
        String[] outFiles = outFile.split("/");
        outFile = outFiles[outFiles.length-1]; //removal of path
        writer = new PrintWriter(outputDestination+outFile+".arff");
    } catch (FileNotFoundException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }

    setup();
    readAndWriteData();
    close();
}

private static void readAndWriteData() {
    while (reader != null && reader.hasNext()) {
        String data = getFormat(reader.nextLine());
        writer.println(data);
    }
}

private static String getFormat(String nextLine){
    if (nextLine == null)
        return "";
    if (nextLine.contains("Website: ")){
        String[] tmp = nextLine.split(" ");
        website = tmp[1];
        if (reader.hasNext()){
            return getFormat(reader.nextLine()); // recursive away the first row
        } else{
            return "";
        }
    }

    StringBuilder sb = new StringBuilder();
    nextLine = nextLine.replaceFirst(""," ");
    String[] info = nextLine.split(";");
}
```java
int length = Integer.parseInt(info[0]);
String direction;
if (length < 0){
    direction = "OUT";
} else{
    direction = "IN";
}

int delay = Integer.parseInt(info[info.length-1]);
if (lastDelay == -1){
    lastDelay = delay; // only does this on start
}

int tdelay = delay - lastDelay;
lastDelay = delay;

sb.append(website);
sb.append(",");
sb.append(globalCounter);
sb.append(",");
sb.append(direction);
sb.append(",");
sb.append(Integer.parseInt(info[0]));
sb.append(",");
sb.append(tdelay);

globalCounter++;

return(sb.toString());
}

private static void close () {
    reader.close();
    writer.flush();
    writer.close();
}

private static void setup () {
    // Setup of static part of file
    writer.println("@relation tor-fingerprint");
    writer.println(); // extra space
    writer.println("@attribute WEBSITE string");
    writer.println("@attribute ID INTEGER");
    writer.println("@attribute DIRECTION \{IN,OUT\}");
    writer.println("@attribute LENGTH INTEGER");
    writer.println("@attribute DELAY INTEGER");
    writer.println(); // extra space
    writer.println("@data");
}
```