Report
BitSmuggler - Bittorrent camouflage for Tor traffic

Dan Cristian Octavian

June 17, 2014

Abstract

We have now lost something we once took for granted: our privacy. If tools were built to take it away from us, we will build tools to take it back.

BitSmuggler provides a solution to the problem of bypassing a censorship and surveillance network firewall by using BitTorrent traffic as a covert channel. It is developed as part of the Tor project to be used in conjunction with the Tor anonimity network in order to allow people subjected to Internet censorship and surveillance to access the web unrestricted and without compromising their privacy.

The project shows that the previously univestigated BitTorrent protocol works well for transporting concealed data over an untrusted network under surveillance. It uses a state-of-the-art cryptographic protocol combined with steganography to create an undetectable communication channel, assuming it is facing a state-level adversary.

A proof-of-concept Haskell implementation of the system was created, which will be further developed to be a part of the Tor censorship-circumvention transport protocols.

Acknowledgements

I would like to thank my supervisor, Professor Susan Eisenbach, for helping me throughout the project and for supporting me to go to the CCS Berlin security conference where I met the people working on Tor and realized that Roger Dingledine knew who I am before I knew who he is.

My gratitude goes to Dr. George Danezis for his substantial contributions to this project and for being really supportive along the way.

My thanks go out to Dr. Herbert Wiklicky, for his rigurous feedback and advice on my project, and for making me think twice before I start running a Tor exit node back home.

I would also like to thank my friends. My thanks go to Dan Demeter for helping me make sense of my network and crypto issues. My thanks go to Julian Sutherland for bringing Haskell into my home and my heart.

Todo list

Contents

Chapter 1

Introduction

1.1 Problem

This is your standard terminal output for network connections in China.

```
> curl http://www.twitter.com
curl: (28) connect() timed out!
> tor --hush
Jun 17 09:10:32.000 [warn] Problem bootstrapping. Stuck at 5\%: Connecting to
    directory server. (Network is unreachable; NOROUTE; count 1;
    recommendation warn)
```

Tor connections fail. Worst of all, twitter connections fail. The connections are blocked by a censorship firewall.

This project shows a novel method to bypass this censorship firewall and avoid network surveillance.

Since 1988, China has been operating a censorship and surveillance system, codenamed Golden Shield which censors Internet access and observes network connections. Similar situations can be found in Iran or Syria. These states resort to building censorship and surveillance firewalls, monitoring and manipulating communications between endpoints in their country and the outside.

The problem at hand is to bypass these censorship and surveillance systems, to allow people affected by them to access the internet freely and not get caught in the process. Simple solutions such as proxies and VPNs are no longer viable and more sophisticated methods are required.

National level censorship and surveillance systems are becoming increasingly sophisticated, not only blocking certain Internet resources but also blocking the use of anti-censorship systems, as shown in the example above [Winter, 2013a]. The real-world

adversary faced by anti-censorship systems is one who passively scans all network traffic passing its national borders inspecting not only network packet headers but also the payload, looking for patterns and performing statistical analysis. It also does active scanning looking for Tor nodes that provide ways to bypass its firewall.

The only remaining way of reliably bypassing these firewalls is to use proxies that are not blocked and obfuscate your network protocol so that it cannot be recognized fast enough by the censor to be blocked. The Tor project is at the forefront of research in this domain and set of solutions addressing this problem are currently being developed ([Project, 2011c]), but no definitive solution has been found yet.

1.2 Contributions

BitSmuggler provides a solution to the problem of bypassing a censorship and surveillance network firewall by using BitTorrent traffic as a covert (camouflage) channel. It is developed as part of the Tor project to be used in conjunction with the Tor anonimity network in order to allow people subjected to Internet censorship and surveillance to access the web unrestricted and without compromising their privacy.

Th1e contributions of this thesis are the following:

- assesed the effectiveness of the BitTorrent protocol as a covert channel and found it fit for the task (design in section 5.2 and evaluation in section 7.1.2)
- developed a cryptographic protocol and a steganography strategy to make a communication tunnel embedded in BitTorrent traffic that cannot be detected by a state-level adversary with certain defined limitations (design 5.3 and evaluation 7.1.3)
- built a proof-of-concept implementation that showcases the performance characteristics of such a system. The system has a high-throughput communication channel with a large factor of steganographic expansion (wasted data to useful data ratio for the sake of undetectability is high) (implementation 6 and evaluation 7.2)

1.3 The idea

The core idea of the project is to create a concealed bi-drectional communication channel between 2 agents by creating a BitTorrent file-transfer connection between them and replacing the transfered file data with their hidden messages. For an adversary eavesdropping on their network connection the traffic would look like normal BitTorrent traffic. The messages between the agents are concealed in the BitTorrent protocol messages through the use of encryption and steganography.

Using Bittorrent traffic as a covert channel is a promising choice since it is one of the most prevalent protocols in the world in terms of traffic volume [**?**]. it has the property that it has a large throughput both downstream and upstream and transports content fit for steganography (eg. video files). Its popularity makes it unlikely to be a target of blocking and doing so would be considered an extreme measure.

The idea to use BitTorrent traffic as a covert channel was proposed by Dr. George Danezis, the author decided to investigate this idea, and thus the BitSmuggler project began.

Compared to other anti-censorship protocols and systems, BitSmuggler stands out through the fact that it chooses a covert protocol never explored before which, unlike other covert protocols, has a very high volume traffic pattern allowing for high throughput for the traffic it conceals.

Furthermore, it is the only other pluggable transport except for CodeTalker tunnel ([**?**]), a reincarnation of the SkypeMorph project [Mohajeri Moghaddam, Li, Derakhshani, and Goldberg, 2012], that uses the original software associated with the immitated protocol in order to create a traffic pattern as authentic as possible. Houmansandr [Houmansadr, Brubaker, and Shmatikov, 2013] argues that is is a necessary design feature if you want to generate cover traffic indistinguishable from genuine traffic.

The fact that it works with a plain-text unecrypted protocol (BitTorrent) means it will work with censors that are very likely to block all encrypted traffic, such as Iran [phobos].

All these features combined make it a unique endeavour and a robust solution to the problem at hand as we shall see in the next chapters.

1.4 Report Outline

The report contains the following sections:

- Background reading

  This section explains the basics of how the Tor anonimity network works and how the anti-censorship protocols integrate with it. The real-world attributes of adversaries responsible for censorship firewalls are discussed based on empirical studies . A quick description of the BitTorrent protocol is provided.

- Security goals and threat model

  These sections present the goals that the project has in terms of security and the assumed adversary model against which BitSmuggler is designed to work. These 2 sections are the pillars for justifying the design choices and for performing the theoretical analysis of the project.

- Design

The design section describes in detail how BitSmuggler works from how the cover traffic is generated and used to the custom data transfer protocol created and the cryptographic protocol designed especially for BitSmuggler .

- Implementation

  The implementation section talks about the proof of concept implementation covering the software architecture, the choice of tools, programming language and libraries.

- Evaluation

  The project evaluation is 2-fold: a theoretical evaluation in the form of a series of arguments as to whether or not the security goals are met given the assumed adversary model; a practical performance evaluation of the proof-of-concept implementation.

- Conclusion and future work

  The last chapter sums up the outcome of the project and talks about the future of BitSmuggler .

Chapter 2

Background

This section covers the information needed to understand this project, assuming no prior knowledge about anonymity networks, censorship circumvention schemes, P2P protocols.

The report assumes general knowledge about networks and cryptography (public key cryptography, symmetric key cryptography, Diffie-Hellman key exchange).

I am going to discuss the following topics:

- the Tor anonymity network - how it works, what is its purpose
- surveillance and censorship firewalls - case studies of censorship firewalls in China and Iran
- Tor censorship circumvention schemes - what mechanisms does Tor currently provide for defeating the above mentioned firewalls
- the need for a better Tor traffic "cover channel"
- The BitTorrent protocol

2.1 The Tor anonymity network

Tor is a an anonymity network and a software that allows its users to defend against traffic analysis, a network surveillance technique meant to gather information about a user's online activity. By using Tor, an individual achieves online anonymity. ([Project, 2011b])

2.1.1 Purpose

Why would one need such a service? Suppose you are interested in accessing a website which is blocked by your ISP (eg. Pirate Bay). Alternatively, suppose you are a government worker in an oppressive state and want to play the whistleblower role and contact outsiders to reveal secret information, action which could compromise your personal safety. Tor is an appropriate tool for this kind communication since it allows you to keep your communication anonymous: eavesdroppers will not be able to know who you are contacting over the network and the party you are communicating with is not aware of your real IP address, unless it is willingly disclosed.

What does Tor effectively offer to users? Why does one need more security than that offered by encryption? It is because encryption hides the data payload of internet packets, while the headers of the packets are visible to any eavesdropper in the network. The packet headers reveal source, destination, size and timing. Tor allows a user to hide this information as well. The receivers of the packets themselves do not know the sender identity either.

Understanding adversary capabilities and intentions gives a more clear image of what privacy protection problems Tor is trying to solve. A traffic analyzer may spy by just sitting somewhere between the source and the destination of the package and observe the packet headers to learn about information exchanges between the parties. It may be more sophisticated and observe multiple parts of the Internet and throw in statistical analysis to track the communications of individuals/organizations.

2.1.2 How it works

To understand how Tor works, it is useful to think of it as trying to hide from someone who is chasing you by using a longer convoluted path and covering your tracks instead of just taking a direct route to your destination. Thus, when communicating through Tor, your data follows a random path, moving from one Tor network node to another until it reaches its destination.

Data packets travel from source to destination through the Tor network on what is called a circuit. A circuit is composed of a chain of relays (machines which are part of the Tor network). Each relay in the circuit knows only about the relay before it and the one after it in the chain. Thus none of them know the entire path of the data they are transporting from source to destination. A different set of encryption keys is negotiated by the client for each hop in the circuit so that no relay in the chain can track communications flowing through it. [Project, 2011b]

The figures 2.1, 2.2 illustrate a Tor usage scenario. Suppose that Alice want to contact Bob by using the Tor network, instead of making a direct connection to Bob.

Figure 2.1 shows how Alice learns about the Tor network from a public directory server

(Dave). Note how the relays are not *secret*. What machines are running Tor relays is public information.

Figure 2.2 shows how Alice builds a circuit of Tor relays to relay its connection to Bob. Data packets travelling from Alice to Bob and back go through each node in the circuit in order.

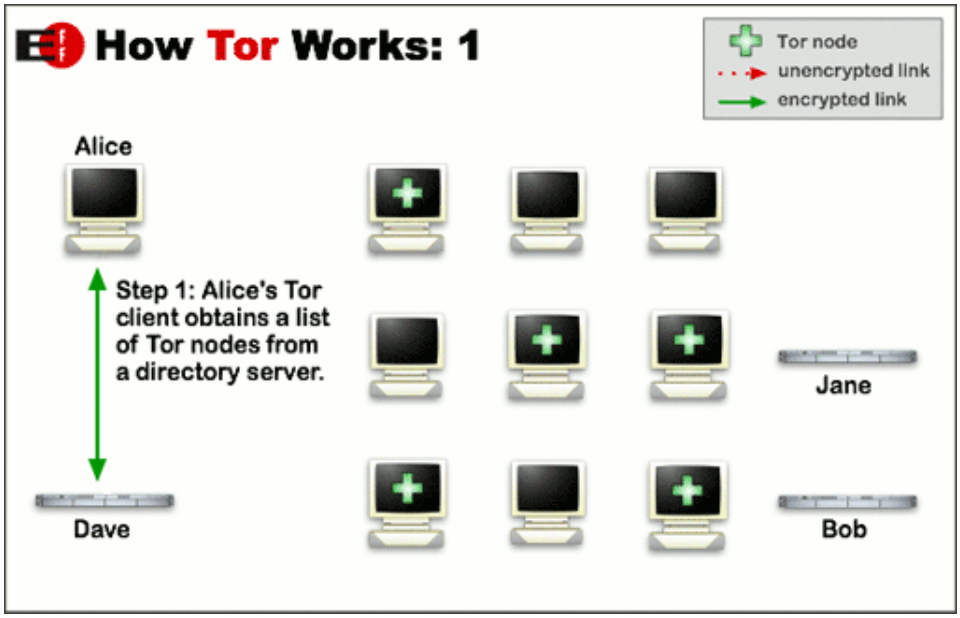

Figure 2.1: A distributed anonymous network; taken from the Tor Project Overview [Project, 2011b]

The above techniques ensure that an eavesdropper sitting between any 2 relays in a circuit cannot deduce any of the information normally given away by headers. Eavesdropping right at the source only reveals that a connection to some Tor relay is made (as shown in the figure 2.2 it is a green Tor encrypted link). Eavesdropping right at the destination reveals a red unecrypted link, meaning the traffic of the user is out in the open.

The exit (final) node

The final node in the circuit is called an exit node. The destination of the Tor user's connection sees the exit node as the source of the connection and is unaware of the real source since it has hopped through the entire chain to get to it. This way, user anonimity is achieved.

It must be noted that the exit node, however, knows all about the user's connection: its destination, its protocol, its headers and its even its contents if they are not encrypted

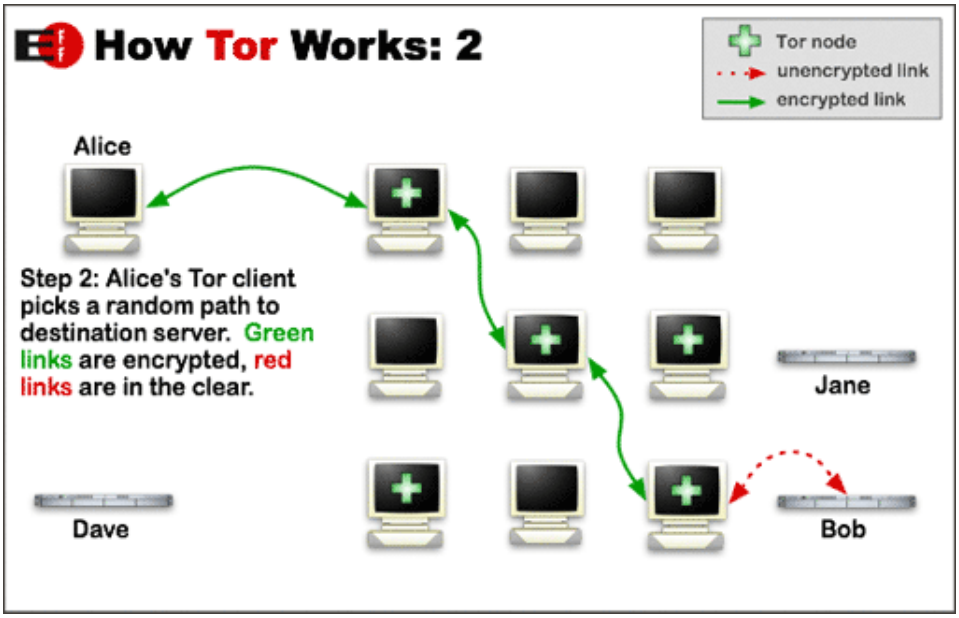

Figure 2.2: A Tor circuit; taken from the Tor Project Overview [Project, 2011b]

(eg. it is using http instead of using https). The only thing it doesn't know is the real source of the connection: who the user creating it actually is in terms of its IP address.

Who runs Tor

The nodes which compose the Tor network are run entirely by volunteers and the diversity of these volunteers is another aspects which works to enforce the privacy guarantees of Tor. Since Tor nodes are spread across the globe and belong to a series of individuals/organizations and every communication through Tor follows a circuit composed of multiple relays, it is very difficult for one entity to be able to monitor the users of Tor.

2.1.3 Issues and potential attacks

Traffic analysis

However, one can argue that Tor security guarantees may be compromised by an adversary who owns a large amount of nodes in the Tor network or who monitors traffic going into and exiting the network to make statistical correlations. This is an acknowledged weakness of the system and it is a an accepted fact that Tor cannot face a state-level adversary with large resources and great reach. Despite this Tor has a large array of use cases.

For example, using Tor to run a criminal organization and hide from the US government

is an approach likely to fail due to their computational and monitoring capabilities. An adversary such as the US has the resources to analyze the network and the influence necessary to monitor the Internet traffic in a variety of places, perhaps outside its national borders.

On the other hand, using Tor to access discuss a business strategy with your busniess partners in order to ensure it is kept secret from your competition is a valid use case, since the adversary faced here is a less powerful one, a company.

Much research has gone in evaluating the effectiveness of traffic correlation attacks on the Tor network, which can be put to use as described above. The most recent work by Johnson on the subject is the most authoritative ([Johnson, 2013]) and it reveals that traffic analysis is more potent than previously believed. The authors develop a framework for analyzing the vulnerability of Tor against attackers owning IXP coalitions or/and being part of the actual network and they show how adversaries with plausible amounts of control over this kind of infrastructure can very effectively detect which parties are communicating using the Tor network. Thus they compromise the anonimity and unlinkability (linking communication source to destination) of the Tor network.

Malicious exit nodes

As discussed earlier, exit nodes are the final nodes in Tor circuits being able to see the user traffic in plain. Of course, the traffic in the clear may have its content encrypted (https) but its headers are now visible.

The issue with this is that the exit one provides its owner with the perfect setup for performing man-in-the-middle-attacks (MitM) as shown by Winter and Lindskog ([Philipp Winter, 2013]). Such possible attacks are DNS poisoning or SSL-based attacks (HTTPS MitM and sslstrip). Furthermore, owners of exit nodes can be annonymous if they choose so. Winter and Lindskog showed empirical evidence of such attacks being conducted.

2.1.4 Tor as a censorship circumvention tool

An interesting use case of Tor is that of using it a censorship and surveillance circumvention tool in countries where Internet access is subject to surveillance such as China and Iran. One could argue that this is a situation where the Tor network is facing a state level adversary and the effort is futile. However, these states tend to be isolated islands to the rest of the world which opposes their policies. This kind of adversary is limited to performing traffic analysis on all incoming and outgoing national traffic, having not much reach outside its borders. Therefore the problem of using Tor to bypass surveillance is reduced to that of finding a way of penetrating their firewall.

2.1.5 Tor bridges

A central element of Tor censorship circumvention tools is the Tor bridge [Project, 2011a]. A Tor bridge is a Tor relay with one essential difference: it is *not publicly listed* in the Tor directory server (figure 2.1).

This feature makes the Tor bridge harder to block by an adversary, unlike a normal relay. Any censoring adversary who wants to prevent the use of Tor could trivially block access to all Tor relays advertised by the Tor directory server based on their IP.

That is why there exists a set of Tor nodes, the bridges, which are not publicly available and their **Bridge descriptors** (records containing all of the Bridge's contact details: address, public keys) can only be obtained through other methods such as providing proof life or communicating with the owner of the bridge. For example, one way of obtaining a Tor Bridge descriptor is by sending an email with a certain format to an email address owned by the Tor project from a Gmail email address. This relies on Google's ability to prevent users from automating email account creation.

2.2 Surveillance and Censorship firewalls

The problem at hand is that of developing a mechanism for allowing people to bypass internet surveillance and censorship systems, as part of the Tor project. This means that such a mechanism aims to defeat a black-box adversary whose resources and techniques are unknown but can be intelligently estimated/guessed. In this section we will go over the necessary background knowledge about this adversary, focusing mostly on China, the owner of the most sophisticated Internet censorship system in the world ([OpenNet, 2013]), and briefly discussing others such as Iran.

2.2.1 Observations

The adversaries aim to inspect information flowing in and out of the state they are operating in, tamper with it and block it as they see fit. They are interested in seeing which are the parties involved in the information exchange, and even though the actual content may be protected by encryption, data about source, destination, timing, size is easily accessible. Furthermore, they have passive and active systems which aim to defeat surveillance and censorship circumvention mechanisms, such as Tor. All of this is achieved by China with the Golden Shield Project (also known as the Great Firewall of China - GFW). ([hikinggfw.com, 2013])

For the purpose of censorship, adversaries employ a series of methods ([Jonathan Zittrain, 2003]):

- filter based web server IP address - access to certain IP addresses is denied. All IP based protocols are affected. This is implemented most likely with block lists loaded onto border routers
- filtering based on DNS IP address - access to DNS servers with certain IPs is denied. May be easier to circumvent if the user has the IP of his target webserver and does not require a name resolution. Most likely uses the same implementation as above.
- DNS redirection - DNS servers in China resolve domain names to web server addresses other than the official ones ([Lowe, 2007])
- filtering based on keywords in the URL - requests are blocked whenever the URLs contain certain keywords. This is most likely implemented with packet-filtering systems integrated with the border routers.
- filtering based on keywords present in the HTML response - packets are blocked based on the content of the response. This is a strong argument for the presence a packet filtering system.
- explictly blocking tools for circumventing censorship: VPNs ([Charles Arthur, 2012]) and Tor ([Winter, 2013a])

Countering censorship circumvention strategies

It was shown that the GFW employs techniques that may be easy to circumvent but the circumvention will not go unnoticed - which highlights an important goal of this project: bypassing the firewalls should go unnoticed. Work by Clayton ([Clayton, 2006]) features an experiment which proves that GFW attempts to kill an undesired connection by sending forged TCP reset packets both ways. The authors were speculating that it uses an offline packet analysis system to decide whether a connection is to be dropped or not. Both parties could theoretically choose to ignore the reset message and continue the exchange. However, the surveillance system could easily log this event and pursue an investigation on the communication endpoint located in China.

To circumvent the censorship methods, the most natural tactic is to use a VPN or proxy. However, this is no longer a viable alternative. VPN connections are often now blocked ([Charles Arthur, 2012]) which proves GFW is detecting and then killing connections to VPNs or proxies ([**?**]). Circumvention tools based on proxies (Ultrasurf, Psiphon) now need to rotate IPs constantly since their servers are continuously blocked. It is very likely that the Golden Shield Project developed tools for detecting VPN, SSH, TLS/SSL traffic and also explicitly analyzes circumvention tools in order to fingerprint their traffic so that they can block them.

According to the work of Winter ([Winter, 2013a]) the GFW is taking strong measures to prevent the use of Tor by its citizens. This is the most extensive study on the issue of

Tor blocking in China. Apart from the most obvious measures such as blocking all public relay addresses and all web servers associated with the Tor project, GFW is blocking the Tor bridges as well. For example, the pool of bridges which are obtained by providing proof of life (captcha solving) on a Tor website is blocked entirely.

Moreover, their experiment, which involved pretending to be a tor client in China, showed GFW is actively probing machines on the web to see if they are Tor bridges. The authors conjectured that DPI (deep packet inspection) boxes are being used to discover packets that are potentially tor packets. Suspect network connections get their addresses placed in a queue. These addresses are later automatically scanned by connecting to them through the Tor protocol. If the machines respond as a Tor bridge would, they are blocked.

Based on the study of the IPs of the Tor bridge scanners, the authors were lead to believe that an IP spoofing scheme is used to hide the origin of the scanners. Therefore, any attempt to distinguish between an honest IP and a scanner IP based on the address alone is unlikely to succeed.

The speculated surveillance systems described above seemed to have flaws and peculiarities: some Tor relay directories were not blocked, some tor bridges never got blocked or got unblocked after a while after exhibiting tor activity. Nonetheless, an intent to minimize collateral damage was observed, since undesired connections were dropped for (address, port) tuples meaning IPs weren't blocked entirely. My belief is that in searching for a solid solution, one cannot rely on these small failures or indulgences. They can be exploited for short term gains, but a real solution to the problem should assume an adversary which has the above capabilities perfected.

2.3 Existing solutions

This section covers the existing techniques for solving the problem at hand, discussing the ones provided by Tor, companies and other open initiatives.

Since traditional VPN/proxy based solutions are quickly becoming obsolete due to the adversary blocking such connections, more sophisticated systems have been developed using techniques various techniques such as large numbers of rotating changing proxies, hidden proxies shared through more secure channels (Tor bridges) and traffic obfuscation/camouflage.

2.3.1 Private company solutions

Censorship circumvention solutions developed by companies deserve attention, even though they are often overlooked in the papers from the research community. The ones worth noting are Ultrasurf and Psiphon. Both companies developed systems which

expand on the traditional model of using a proxy, but throw in techniques such as rapidly rotating IP addresses of their proxy servers to a ensure there is a set of unblocked proxy servers at all times and some connection obfuscation techniques.

Both Ultrasurf and Psiphon ([ultrareach.us, 2014]) ([psiphon.ca, 2014]) employ a fleet of proxy servers, and bootstraps the client software with a set of addresses of proxy servers, allowing for further discovery of other proxies as the old ones get blocked. Psiphon aims for a simplified client experience, offering also a no-installation solution through a web app for the sake of leaving no trace on the user's machine. Ultrasurf states guarantees about the absence of traces as well. Psiphon's more interesting feature is its traffic obfuscation technology. It features multiple channels for transmitting data: VPN, HTTP, ssh and obfuscated ssh, switching from one to another in an attempt to find one that is not detected by the firewall based on the user's use case. It It argues its effectiveness based on its popularity claiming 260 million + pages have already been served using siphon. Ultrasurf boasts about supporting traffic worth of millions of pages daily.

Both these services are closed source and lack transparency, while one of them has been shown to have serious issues. Ultrasurf has received the attention of security expert Jacob Appelbaum who present a study [Appelbaum, 2012] about his analysis of Ultrasurf through reverse engineering and discussions with the developers. According to his findings their software has numerous flaws, from non-up to date servers thus vulnerable to external attackers, lack of forward secrecy and overstated effectiveness (their proxy servers are often blocked in censoring countries) and their easy to filter bootstrapping process. Thier log retention policies are dubious at best, and they acknowledged to have submitted logs to the US government.

My take on the case of these companies and their services is that their model has inherent flaws. The closed source nature of these project means the entire responsibility of the software's security is taken up by the company itself, since no research community can peer review it, this being an unwritten standard in the security world. Secondly, they act as single hop proxies and most likely keep logs, meaning that if an attacker gains control of a proxy he can see both source and destination of all traffic flowing through it. Moreover, none of them employ any solid method for avoiding their traffic being fingerprinted and easily blocked.

2.3.2 Tor solutions

In the context of the Tor project, research has been focusing on finding ways of hiding the Tor traffic from filtering and on avoiding tor bridge blocking. These efforts came as a direct response to the effort of GFW to filter Tor traffic through DPI and perform active probing on Tor bridges.

The general approach of the Tor project has been to favour the easy development of surveillance and censorship circumvention mechanisms in order to obtain a set of reli-

able solutions which can be used interchangeably based on what works best in a certain situation. Thus, the project now encompasses Obfsproxy, a framework for developing pluggable transports, traffic transformers meant to make the Tor traffic undetectable. The most notable pluggable transports at this moment are ScrambleSuit, StegoTorus, Skypemorph, Dust and Format-Transforming Encryption ([13] Tor project, 2013). Flashproxy is another popular tool in the anti-censorship arsenal, which does not provide a cover for the traffic but offers a solution to the bridge blocking problem.

Proxy blocking solutions

Even though the focus of this project is to develop cover traffic for Tor, it is useful to look at Flashproxy ([14] Fifield, 2012) as it can combined with existing filtering circumvention techniques. The main idea behind it is to generate many ephemeral proxies for Tor traffic so that blocking them is a large effort which is almost pointless since they are short lived and quickly replaced by others. The neat trick used to achieve this is relying on random internet users to become proxies for a short time by running a Javascript script in their browser which proxies traffic from Tor clients to Tor relays. The script can reach a user's machine by being embedded in a page served by a collaborating website.

Traffic filtering circumvention

Most traffic filtering avoidance projects rely on the idea of camouflaging traffic under other types of traffic, and sometimes making use of steganography to hide the payload. Some rely on creating look-like-nothing traffic. Most of the ones that are using camouflage are simply mimicking the target protocols and this is shown to be a flawed approach in the paper: "The Parrot is Dead: Observing Unobservable Network Communications"([15] Houmansadr, 2013). We shall discuss each in turn and highlight their good and bad parts.

Skypemorph ([16] Moghaddam, 2012) is classical example of a pluggable transport that uses camouflage to avoid detection by packet filtering systems. The idea behind it is simple: use a Skype connection as a covert channel for Tor traffic and send Tor packet data instead of voice data. The implementation of this system was from simple though: the authors took an approach through which they combine the use of the Skype client for handshakes and a separate application which mimics the Skype protocol but instead sends Tor data packets. This choice of cover channel provides high throughput but the task of imitating Skype and, in general, any fairly complex protocol, is non-trivial and prone to errors. We will see further on how this is potentially a great vulnerability.

Stegotorus ([17] , 2012) acts as a framework for developing pluggable transports which make use of steganography. It provides a novel way of encryption fit for use with steganography, a generic architecture meant to fit any steganographic cover protocol

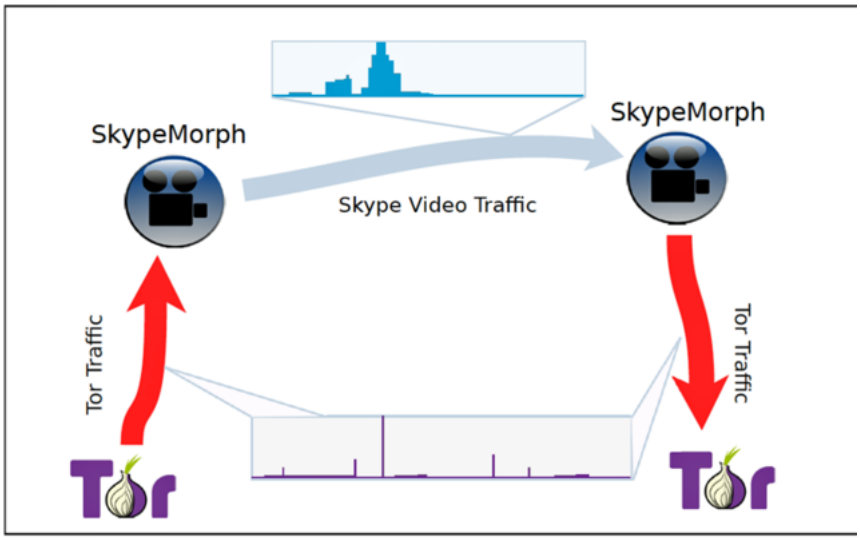

Figure 2.3: How skypemorph tunnels Tor traffic over a censored network

and 2 proof-of-concept steganography modules. It also provides a systematic way of attacking their pluggable transport to demonstrate its resilience. The ideas behind the novel encryption scheme, the framework and the attacks are good, but the steganography modules are vulnerable to trivial fingerprinting as shown in the paper "The Parrot is Dead". Furthermore, the bandwidth of the cover channel using a HTTP steganography module is quite low (30kb/s).

Scramblesuit ([18] , 2013) takes a different approach from the aforementioned and does not attempt to mimic an existing type of traffic and creates look-like-nothing cover traffic which looks random, in line with the idea that a imitating existing types of traffic is likely to fail. They pay great attention to their traffic shape so at to make it non-fingerprintable. Their approach also has much better network throughput than others. The one great weakness of the authors' approach is that their tool will be completely blocked off by a whitelisting censor, since their type of traffic is classified as unknown.

Dust ([19] Wiley, 2013) provides a packet based DPI resistant protocol, as opposed to connection based. It acts as an engine for generating protocols to defeat filtering. It makes packets follow a certain format (perhaps based on an existing protocol) and makes packet size follow a defined distribution. However, unlike scramblesuit, it does not take into consideration packet interarrival times. Nonetheless, it allows for something clever, namely configuring the statistical properties of the encrypted content in order to avoid attacks such as measuring entropy which could detect normal encrypted text by its high entropy.

"The parrot is dead: Observing Unobservable Network Communications" is a clever paper which finds flaws in existing proposed filtering circumvention systems. It managed to find efficient and simple ways of fingerprinting Stegotorus modules, Skypemorph and

Dust through of combination of packet inspection and active probing. The point of the paper is to state that imitating cover protocols is doomed to fail and that a better option is to ride on "real" traffic generated by the software using the protocol. I believe they make a valid point and all future camouflage transports should take this into consideration.

Need for better protection against traffic filtering

As we have seen from the discussion in the previous sections, there is no robust long-term solution at this point for avoiding traffic filtering and "smuggling" tor traffic unnoticed through the Firewall of a competent adversary. It is true that many of the current solutions will work for the current state of things, but further research has shown that if the attacker improves his DPI algorithms and combines them with active probing he can defeat them. Furthermore, the idea is to develop a set of such solutions and choose which fits best each use case.

It is important to point out that this is a continual arms-race and future solutions are unlikely to be a silver bullet. Any solution that puts us in front of adversary or forces the adversary to invest large amounts of resources and time is a good one. For example, if a circumvention strategy requires the adversary to do massive infrastructure upgrades to counter it - consuming time and money, it is an important step forward.

2.4 The BitTorrent protocol

The BitTorrent protocol is peer-to-peer file sharing protocol presented by its creators as a free speech tool [bittorrent.org, 2014]. It allows a user to publish and distribute files to other users.

It is an extremely popular protocol and in 2012, 2 of its most popular clients BitTorrent Mainline and uTorrent had a userbase of more 150 million people [**?**]. It is responsible for 30% or more of the Internet traffic volume in the US, Europe and Asia accordinng to the records published by Sandvine [**?**].

The swarm

The protocol [**?**] is a scalable solution to the problem of data distribution. Given that a set of users requires a set of files, instead of using a client-server model to provide clients with the file, the file is shared from client to client in a cooperative manner.

A group of clients sharing a particular file or group of files is called a **swarm**.

The members of a swarm can be either:

- **peers** - clients that don't yet have a complete copy of the file

- **seeds** - peers who have complete copy of the file; they remain in the swarm as file uploaders only

The metainfo file

The file being dowloaded has a corresponding descriptor file, called a **metainfo file** usually marked with the extension .torrent. This descriptor is *necessary* so that the BitTorrent client can perform the download. The torrent descriptor contains metadata: it specifies how the file is split in **pieces** and provides a SHA1 hash of each piece so that data integrity of the downloaded file can be checked. It also contains a global torrent hash which uniquely identifies the whole file based on its content.

The entire file is split into pieces of varying sizes to make distributed sharing easier.

Traditionally, the protocol operates with another party involved: the **tracker**. The tracker is an actual server that keeps track of each torrent descriptor and the swarm associated with it. When a client seeks to download a certain file it queries the tracker mentioned in the torrent descriptor. (figure 2.4)

Lately, the protocol has shifted away from the tracker model towards a completely distributed design. All the user needs is the hash of the file that it wants to download, called a **magnet-link**. Based on a system called **DHT** (Distributed Hash Table) and querying with the magnet-link as the key, it is able with find out with the help of other BitTorrent clients which is the swarm working on the desired file and obtain the metainfo file. The only place where a server might be involved but *not necessary* is in the distribution of magnet-links (piratebay) and in the initial bootstraping of the BitTorrent client with other BitTorrent client addresses so that DHT can work.

Peer communication

The peers communicate using a *single* connection that is symmetrical. Messages that flow in both directions look the same. The messages are either control messages (eg. chocking/unchoking the connection - meaning more/no more data should be sent, piece requests) or data messages called **piece messages** containing **blocks** of the file being transfered. Blocks ar a subdivision of a piece that are defined by the piece's index, the offset in the piece and their length.

Torrent clients

Torrent clients are pieces of sofware used to download files. They all implement the core protocol while some implement extensions improvements to the protocol. For the purpose of this project it is important to note the relevant properties of the most popular BitTorrent clients in use([**?**]) in table 2.1.

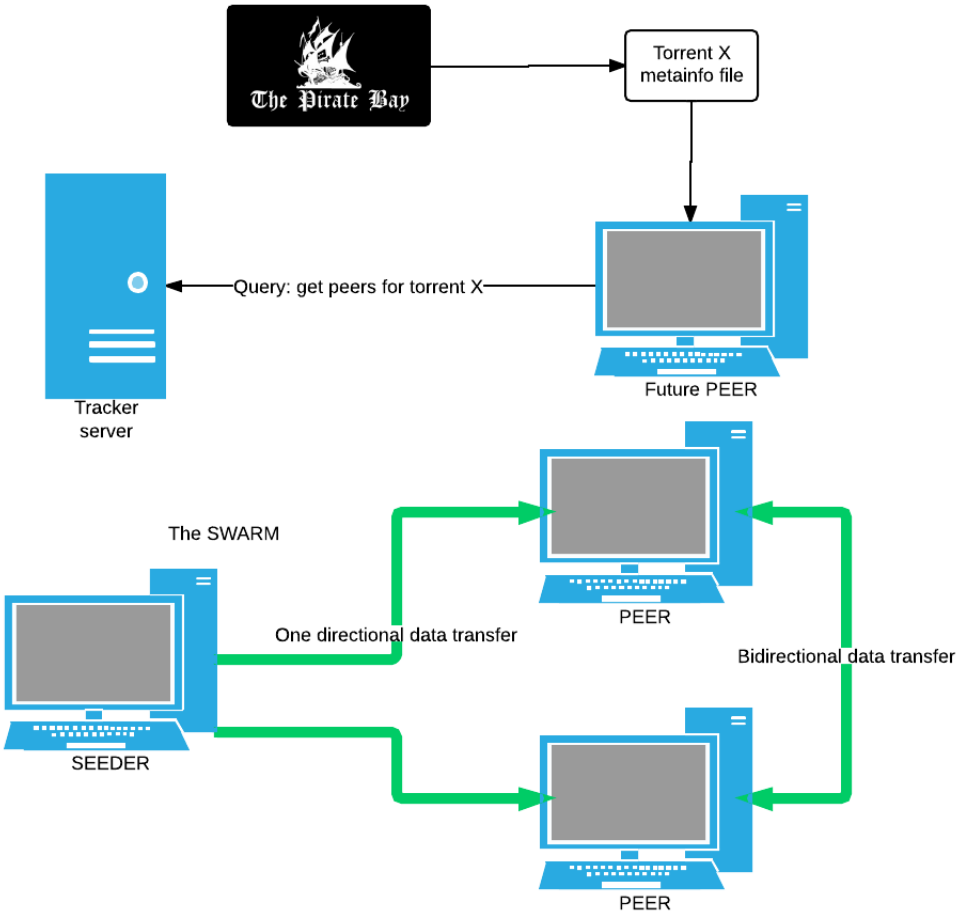

Figure 2.4: Traditional bittorrent architecture: peers learn about the swarm from the tracker

Since BitTorrent clients will be a part of the system being build we present the relevant characterstics along with their usage share.

The usage share data was compiled based on a study by Delft University. Note, however, that is not particularly accurate whne it comes down to global usage share since research focused on popular swarms listed on English language BitTorrent sites. An older study from 2009 made by EZTV's NovaKing ([ernesto, 2009]) reveals a more global view, yet more outdated: back then the top BitTorrent client was the Chinese BitTorrent client **Xunlei** with market share of 29.3%.

Client	Usage share	Open source	Platforms	Proxy server	DHT
uTorrent	47.97%	No	Windows, Mac, Linux	yes	yes
Vuze	22.49%	yes	Windows, Mac. Linux	yes	yes
BitTorrent mainline	13.01%	no	Windows, Mac. Linux	yes	yes
Transmission	7.0%	yes	Mac. Linux	no	yes
Libtorrent	1.02%	yes	Windows, Mac. Linux	yes	yes

Table 2.1: Most popular BitTorrent clients

Chapter 3

Security goals

The goal of this project is to create a network communication channel that cannot be subject to surveillance and censorship and integrates with the Tor network, allowing a user to access the Internet anonimously, unhindered and unseen by a surveillance firewall.

3.1 Security requirements in a concrete scenario

The concrete scenario below showcases an individual who would want to be able to access any internet resource without being blocked, without the surveillance mechanism knowing that he/she is trying to get around it and, perhaps he/she would want to use all facilities of the Tor service.

Suppose you are currently in Beijing and want to access your facebook profile to talk to your friends. A direct connection fails immediately. The domain is blocked. You think of alternative ways of hiding your access and you remember you have an openvpn account. When you navigate to openvpn.com you are blocked once again. One free, reliable option is also Tor. So you spin up your Tor browser and attempt once again to connect to facebook.com. The Tor client fails to connect to relays though: all connections are dropped. [blockedinchina]

You can be certain that by now that whichever mechanism is blocking your connections has already logged your attempts to use facebook, openvpn and Tor, your location and IP. All the conventional ways of getting around traffic censorship have failed you, because this is not your standard ISP censoring piratebay.com by default. Your traffic is under surveillance by The Great Firewall of China and conventional ways of getting through it have been take care of. Also, just because your traffic goes through for some web addresses doesn't mean you aren't being watched. The arms race continues.

3.2 Formal definition

In this section we formalize the security goals of the project shown informally with the above scenario.

The project aims to preserve the following Tor security goals:

- Unlinkability -The adversary (censor) should not be aware of what 2 parties are involved in a network communication through Tor
- Performance -The performance of the communication channel should still be close enough to real-time so that the users are willing to accept the tradeoff in the favor of increased secuirty

Aside from this, the project sets its own security goals:

- Undetectability - the adversary should not be able to detect whether a user is using BitSmuggler , its detection capabilities being limited by the computational constraints described in the threat model (4)
- Confidentiality - implied directly by undetectability; content of the communication is not known
- Unblockability - the adversary should not be able to block BitSmuggler without blocking a large amount of unrelated traffic/causing a lot of collateral damage
- Plausible deniability - the user of BitSmuggler should be able to deny its use. An advesary could only prove that the user did use BitSmuggler ony if he inspects his machine. Otherwise any traffic generated by BitSmuggler should pass as a normal BitTorrent file exchange.
- Resist active probing - if the Tor bridge running BitSmuggler is contacted by the adversary in order to check if indeed it is running Tor (active probing), BitSmuggler should avoid giving any sign of the existence of a Bridge. This is assuming the adversary does not have the Tor bridge's descriptor (address, public key) and is just shooting in the dark.

Chapter 4

Threat model

We now describe a model of the adversary that BitSmuggler works against, by stating the *assumptions* made about its capabilities, limitations and intentions.

This adversary model is very similar to the adversary model of the StegoTorus project [Weinberg, 2012] and borrows many elements from it . This is because BitSmuggler is a lot like StegoTorus, who camouflages Tor traffic in common traffic types such as HTTP and uses steganography to hide the data payloads. Some points are inspired by the Telex project [Wustrow, Wolchok, Goldberg, and Halderman, 2011] and SkypeMorph project[Mohajeri Moghaddam, Li, Derakhshani, and Goldberg, 2012] threat models as well.

The adversary (censor) has the following attributes:

- is a state-level authority
- has an interest in connecting to the Internet and allowing the individuals subjected to censorship to connect to parts of the Internet for the socio-economical advantages it provides. It is thus interested in avoiding *over-blocking*
- the state-level adversary has control over a section of the global network which we will call the *national network* delimited by its national borders. We shall call this delimiting perimiter its *network perimeter*.
- has full active and passive control over the national network. The censor can passively monitor all traffic entering and leaving its networks. It is possible for the censor to actively tamper with traffic by injecting, modifying or dropping it or just hjacking TCP sessions
- performs Internet traffic filtering at the network perimeter based on: address filters, content pattern filters, statistical filters; uses Deep-packet-inspection (DPI) to look at the contents of the packets (more on this below 4.1)

- can be suspicious of encrypted network connections (ssh, vpn protocol, https, BitTorrent encryption, tls) and may choose to block them without a strong justification

The adversary's objectives are:

- block access to certain Internet resources
- monitor all Internet usage
  - know which parties are communicating
  - know the content/type of their communication

The adversary's limitations are:

- DPI real-time analysis has strong time limitations for the computations that can be performed while on-the-side anlysis forces the censor to select a very small portion of the traffic for more complex analysis (discussed below 4.2)
- does not have access to the censored users' machines. Alhough the censor may issue mandates to inspect user devices or infect user devices with surveillance software, BitSmuggler cannot service the user reliably if his/her machine is compromised
- has very limited abilities outside its network. It does not control any external network infrastructure or any popular external websites.
- is subjected to economical constraints - improving its surveillance infrastructure to counter new circumvention methods takes time and money
- has strong socio-economic reasons not to block BitTorrent traffic
- does not own enough malicious Tor relays to allow it to land attacks which compromise Tor's unlinkability

4.1 More on traffic filtering

Address filters are filters that look at IP address at other end of the connection and decide whether the connection is allowed or not by keeping track of an address blacklist. For example, all public Tor relays are blocked based on their IP address with address filters.

Pattern filters rely on DPI capabilities to search for certain cleartext patterns in the network packet. One application of this was blocking of ssh connections based on a static pattern found in the plaintext ssh handshake. A counter measure was developed in the form of obfuscated-openssh [brl github repo owner, 2009].

Statistical filters involve looking at patterns of packet interarrival times and size or statistical properties of the payload of the packets. For example, one can attempt to

measure the entropy level of a payload normally containing English text to see if it matches an expected level or it shows a much higher level indicating it may be hiding a steganographied encrypted payload.

4.2 Adversary DPI limitations

In order to apply the above filtering strategies, we assume the adversary is going to perform DPI on the traffic passing in and out of the network perimeter both in real-time and and on the side, by selecting a very small part of the traffic for a more detailed analysis.

The authors of StegoTorus [Weinberg, 2012] make an interesting back-of-the-envelope calculation with regards to the available computation me for each packet inspected. By spending 2 microseconds analyzing each packet passing through the backbone router of Chicago, the throughput of the network is halved. Based on this reasoning, any real-time computation performed on inspected packets is bounded in time to something at the scale of 1 microsecond in order to not impact the performance of the whole national network significantly.

To ballpark how much computation can be done in that time, a C program compiled with g++ -Ofast takes 1 microsecond to do the sum of squares of an array of bytes of size 650 on a 2.2 Ghz Intel ie processor. Thus whatever analysis is performed cannot be much more than a O(N) complexity algorithm on the whole packet (N the size in bytes of the packet).

An example attack deemed feasible with a modern DPI box in real-time by the Stego-Torus paper is one which involves looking at the size of packets of a connection, knowing that a vanilla Tor connection (no plugabble transports) consistently produces packets of a certain size. Through a simple stochastic analysis, they claim they can detect a Tor connection after observing a few dozen packets.

We can deduce from the above that simple constant-time or linear-time statistic analysis attacks are within scope of the assumed adversary.

Support for the existence of on-the-side analysis has been found empirically through experiments which showed how The Golden Shield project blocks Tor bridges through active probing after having seen patterns of Tor traffic in the packets send to and from the bridge's address.

Chapter 5

Design

We now present the design of BitSmuggler and discuss how it achieves its security goals in the face of the described adversary model.

The central idea of the project is to camouflage the Tor traffic as BitTorrent traffic, so that the Tor connection looks like a normal BitTorrent connection in the eyes of the adversary. The project focuses on the specific problem of the camouflaged channel and assumes solutions exists for the problem of how Tor bridge descriptors reach the users of bridges.

We discuss in this section how BitSmuggler embeds the transported Tor traffic in BitTorrent traffic and how it creates an authentic BitTorrent traffic pattern 5.2. We then have a look at how it makes sure that the embedded payloads cannot be spotted through the use of steganography 5.5 and what cryptographic protocol it uses to hide the data it is transporting 5.3. We also show how we resist active probing with a very simple design feature 5.6.

5.1 The design in a nutshell

Two parties, A nd B want to communicate with each other using a secure channel that has the security properties discussed above. Their communication follows a client-server pattern, where A (client) always intiates communication with B (sever). Both A and B each run a BitTorrent client. The BitTorrent clients make a peer connection and start sending pieces of a file to each other.

In order to send messages to each other, A and B occasionally drop their messages in the bidirectional flow of BitTorrent traffic and the messages end up on the other side, carried in the content of the BitTorrent Piece messages. The BitTorrent flow runs unprerturbed by the insertion of messages.

To an adversary, the connection looks like a normal BitTorrent connection. Through a proper choice of cryptographic tools and use of steganography the content of the traffic generated by the connection looks genuine to an adversary observing it. In short, the displaced content of the BitTorrent pieces is compressed media data which looks very much like a random string and the content replacing it looks like random data as well.

5.2 BitTorrent disguise

5.2.1 Why BitTorrent

There are 2 main ways of going about smuggling traffic over a censorship firewall.

Make look-like-nothing traffic

This kind of traffic does not resemble anything already in existance and it is meant to have no distingushing characteristics so that you can identify it with a filtering analysis. A good example of this is the ScrambleSuit project. Challenges:

- generate traffic that cannot be fingeprinted in terms of timing, packet size, content

Main weaknesses:

- it is rendered useless if the censor uses *protocol whitelisting*.

Make it look like another existing protocol

If that type of protocol is allowed by the censor and blocking it off completely would cause a lot of collateral damage, the censor has no choice but to analyze all traffic of that type hoping to find the connections that are not genuine. Challenges:

- make your traffic seem genuine to the adversary
- deal with the performance characteristics and transport guarantees of the protocol you are running under

Main weaknesses:

- you rely on the fact that this kind of traffic will never be completely blocked off
- the efficiency of your system relies on protocol properties that are outside of your control: how popular this type of traffic is, what changes are made to the protocol

BitSmuggler goes for the second approach. The look-like-nothing approach is already being explored by other projects (ScrambleSuit, Dust) while hiding under an exisiting

protocol has a lot of opportunities unexplored. It also does not have the great weakness of becoming useless when the censor uses protocol whitelisting. Thus we bet on banning BitTorrent traffic being less likely than putting a whitelisting mechanism in place.

Given the choice to use an existing type of traffic as camouflage, one needs to find which type of traffic and associated protocol are good options for acting as camouflage. BitTorrent was chosen based on a set of main criteria in mind, enumerated below:

- **popular protocol** - meaning there is a large volume of traffic in the real world using this protocol produced by normal Internet users. This requirement exists to make an adversary's job of detecting non-genuine connections as hard as possible by forcing him to analize as much traffic as possible.
- **high traffic volume** - normally produces a lot of traffic both upstream and downstream. This is because the camouflaging traffic volume sets an upper bound for the camouflaged traffic volume.
- **works non-encrypted** - based on the principle of hiding in plain sight. Encrypted connections are often considered suspicious and blocked off by censors (Iran was blocking all connections which used SSL [phobos])
- **opportunities for steganography** - data payloads can be concealed in the messages of the chosen protocol. The space which the concealed payloads can take add another upper bound to the camuflaged traffic volume.

The BitTorrent protocol has those properties.

It is a popular protocol, according to the Sandvine Internet phenomena report [Sandvine] which studied internet traffic in different regions of the world. In North America, it is the number 1 contributor to the volume of upstream traffic (34.81%) and ranks as the 4^{th} contributor to downstream traffic (5.57%) with an aggregate share of 9.23% (ranking 4^{th}).

In Asia/Pacific, BitTorrent is even more popular as it takes 1^{st} place in the aggregate share with 21.66% of the total traffic volume, ranks 1^{st} for the upstream traffic and 2^{nd} for downstream traffic.

It should be noted Bittorrent has seen a decreasing trend in usage share in North America. The trend is best explained by the rise of Netflix which ranks 1^{st} in the aggregate usage share. Both Netflix and BitTorrent are mostly used for the delivery of media content so one can deduce how BitTorrent lost in favor of the other. It can be argued that this decreasing trend is not a concern in countries with Internet censorship. For example, China has no issues with copyright infringement but it often bans Western media distribution services (eg. YouTube) so BitTorrent traffic is likely to be still very popular there in the future.

BitTorrent optimizes for data throughput since its purpose is to send/receive files and therefore has a large traffic volume per connection.

The vanilla BitTorrent is non-encrypted and to a naive observer it seems like all the data that passess through a connection is in plain-text.

The BitTorrent protocol messages are either control messages or data messages called Pieces. Pieces are file fragments and their data format is whatever data format the file being sent has. There are plenty of opportunities for doing steganography.

5.2.2 Traffic generation

The main challenge in using an existing protocol to conceal other types of traffic is generating genuine traffic that cannot be distinguished from a real connection's traffic.

The approach taken by many projects is to imitate the camouflage protocol with a piece of software that generates similar traffic (SkypeMorph, StegoTorus). However, this approach is very error prone. For any non-trivial protocol, imitating its behaviour exactly is a very hard task. The issue with this is that *any difference* from the original protocol can be used by the adversary to fingerprint the connection as a non-genuine connection as pointed out in "The Parrot is dead" paper [Houmansadr, Brubaker, and Shmatikov, 2013].

The approach taken by BitSmuggler is to use *real* BitTorrent clients to run the BitTorrent connection and produce the traffic. Thus, all issues associated with imitation go away.

The tradeoff is that the following implementation challenges need to be solved:

- a foreign piece of software needs to be integrated in the system - many popular clients are not open source (eg, uTorrent, Xunlei)
- the foreign BitTorrent client opens up a surface for attacks - closed source BitTorrent clients are a potential vector of attack IF influenced by the adversary (eg. Chinese closed-source BitTorrent client Xunlei could have a rogue patch working against BitSmuggler) 5.2.4
- controlling the client is done by sending messages in the local network given some kind of web API
- embedding camouflaged traffic becomes a problem of capturing and tampering with the BitTorrent client's traffic and not as easy as just programatically building BitTorrent messages and altering the content as we see fit
- BitTorrent connections have a complex context: peers are in swarms with other peers and data flows bidirectionally between 2 peers only if both require pieces from the other. This context needs to be created realistically for data exchanges to work 5.2.3

It is a good tradeoff because the task of imitating the protocol correctly is much harder: not only does every corner case needs to be covered but also any changes to existing

BitTorrent clients need to be accounted for. This involves a continous maintenance effort of checking the behaviour of the imitated software and updating the imitator accordingly.

5.2.3 BitTorrent swarm handling

As discussed above, since the system generates BitTorrent traffic using a real client, it needs to create the right context for the BitTorrent client to exchange data with another client in a peer connection.

Two BitTorrent clients to get in a peer connection to exchange a file, they need to be part of the same swarm of peers which exchange a particular file.Furthermore, so that piece messages keep travelling back and forth and allow for bidirectional data transport, the clients need to be in a state in which they keep requesting pieces from each other. (figure 5.1)

The server's BitTorrent client acts as a peer in a predefined swarm.

To create this peer relationship between the 2 communicating sides, the client needs to know the following things:

- what swarm to join to talk to the server given either a .torrent file or a magnet-link
- address of the server, in order to identify which of the peers in the swarm is the actual server
- server public key, to set up an encrypted tunnel

Swarm change

Normally, the 2 peers talking to each other will reach a point at which they exchanged all pieces they could possibly exchange. No data flow will exist after this point in time. Therefore, in order to continue the communication between server and client, they need to change swarms. The server is entasked with proposing a swarm change. In the same way he setup the initial swarm, he can setup a different swarm and instruct the client to switch the connection.

Reconnecting follows the same steps as the initial connection. When notifying the client of a swarm change, it gives the client a *session cookie* which the client can send after it reconnects so that it preserves its session.

Rejected alternative

To create a bidirectional flow of data, another type of BitTorrent relationships were considered: seeder-to-seeder - the server and the client seed each other to send data to each other. Rejected on the basis this is a suspicious state: 2 peers seeding each other is not a frequent occurrence and this would help the attacker narrow down the connections he analyzes or outright blocks.

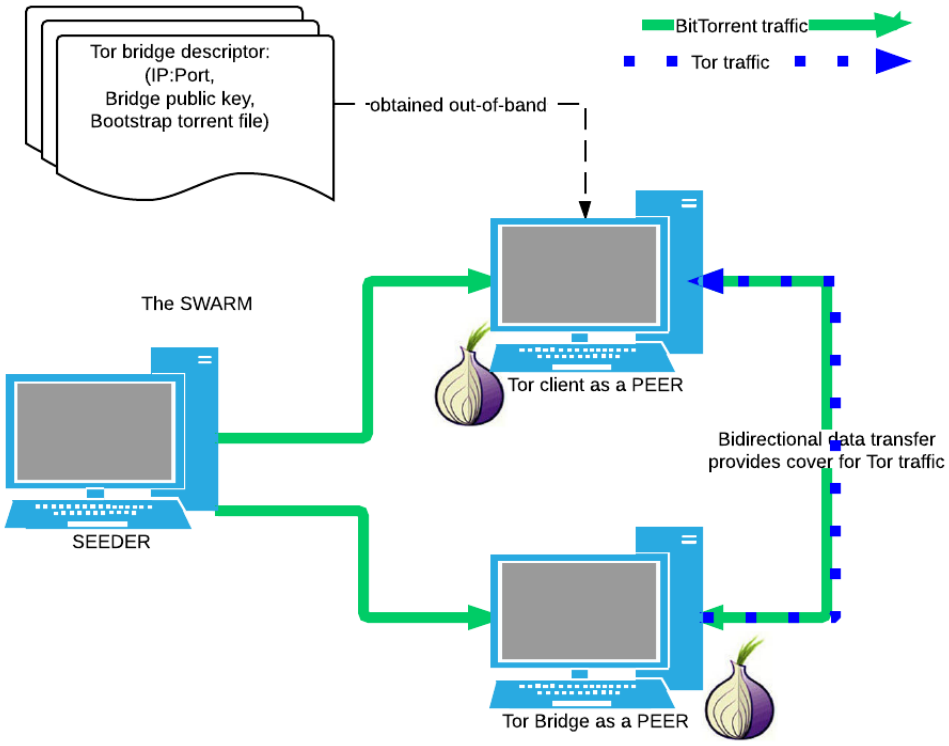

Figure 5.1: BitTorrent swarm used by BitSmuggler client and server

5.2.4 Trusting the Bittorrent client

The BitTorrent client is a component of the system that may be closed-source and coming from a different source other than the Tor project. What happens if the client contains malware used to attack Tor users? A plausible concrete scenario is that some BitSmuggler instances use the closed-source BitTorrent client Xunlei for users located in China, since that is the most popular torrent client in China. Xunlei has been found to contain malware in the past, therefore it is plausible that the state-level adversary may add a rogue patch to it meant as an attack against BitSmuggler .

However, it is stated in the security goals that BitSmuggler is not meant to work when the user machine is compromised, For the given adversary model, this issue is outside of the scope of the project. But since the system itself *requires* the BitTorrent client the possibility of adversarial code placed in the client was taken into consideration. In the case of a closed-source client, no claims can be made about what functionality it is actually running, while open source BitTorrent client binaries should only be trusted if coming from a trusted source.

Solutions to this issue that can be explored in the future are:

- avoid using closed-source BitTorrent clients - the problem is that the most popular clients are closed source (uTorrent, Xunlei)
- sandbox the clients - make them unaware of their context by sandboxing them in a VM.

5.3 Cryptography

The cryptographic protocol of BitSmuggler is designed to guarantee confidentiality and at the same time preserve the undetectability of the communication. The protocol was proposed by Dr. George Danezis.

The protocol makes use of the following tools:

curve25519 Diffie-Hellman function

Curve25519 is an elliptic curve cryptography curve and set of parameters designed to be used with Elliptic curve Diffie-Hellman (ECDH).

The function is used to derive a public key given a private key and to derive a shared secret given that 2 agents know each other's public keys.

Elligator

Elligator is an encoding that makes elliptic-curve points indistinguishable from uniform random strings. This encoding provides you with a 2 functions:

- $generatePublicKey :: PrivateKey \rightarrow (Curve25519PublicKey, KeyRepresentative)$ this function takes a 32-byte private key and generates a curve25519 public key and a uniform random string representation of that key
- $representativeToPublicKey :: KeyRepresentative \rightarrow Curve25519PublicKey$ this function take a uniform random string representation of a Curve25519 public key and transforms it into the actual key

Elliptic curve points are points on a curve defined by a mathematical function and they have mathematical properties based on which they can be distinguished from a random string of bytes. Elligator is necessary so that the presence of an eliptic curve point in plain-text in a stream of random-looking data is hidden.

Assumptions

It is *assumed* that the client knows the public key of the server P_S, because it obtains out-of-band the Tor bridge descriptor containing this key. It is assumed that all the information in the bridge descriptor can be trusted to be correct.

Protocol steps

There are 2 parties communicating: a client (C) and a server (S). (5.2)

The server generates a 32-byte private key K_S and derives a public key $P_S = \text{curve25519}(K_S, 9)$. The server then publishes its key in its Tor bridge descriptor.

The client acquires the publick key P_S out-of-band.

The client generates a 32-byte private key called K_C and derives a curve25519 public key P_C and a uniform string representation of it U_C with $(P_C, U_C) = \text{Elligator.generatePublicKey}(K_C)$.

The client then generates the shared secret between the server and the client K_{SC} with $K_{SC} = \text{curve25519}(K_C, P_S)$.

The client initiaties the communication. By sending U_C to the server in plain-text over an untrusted-channel. U_C has the appearance of a uniform random string, and thus its presence cannot be detected by an adversary, preserving undetectability. Along with U_C, the client sends a handshake message to the server $M_{handshake}$ enrypted with the derived shared secret K_{SC} as such: $\{M_{handshake}\}_{K_{SC}}$.

Once the server receives U_C, it computes the curve25519 public key P_C by using the function $P_C = \text{Elligator.representativeToPublicKey}(U_C)$.

Then the server derives the shared secret between the server and the client K_{SC} using $K_{SC} = \text{curve25519}(K_S, P_C)$.

It then decrypts the handshake message $\{M_{hanshake}\}_{K_{SC}}$ using the shared secret.

The server S now sends an acknowledgement message M_{ACK} to the client encrypted with the shared secret K_{SC} as such $\{M_{ACK}\}_{K_{SC}}$

At this point, both the client and the server know the shared secret K_{SC}. From this point onward all messages sent between the sides are encrypted using symmetric key encryption based on the shared secret.

5.3.1 Low level cryptographic details

To perform symmetric key encryption of the messages sent between the server and the client, **AES in GCM mode** is used as mode of operation for the cryptographic block cyphers.

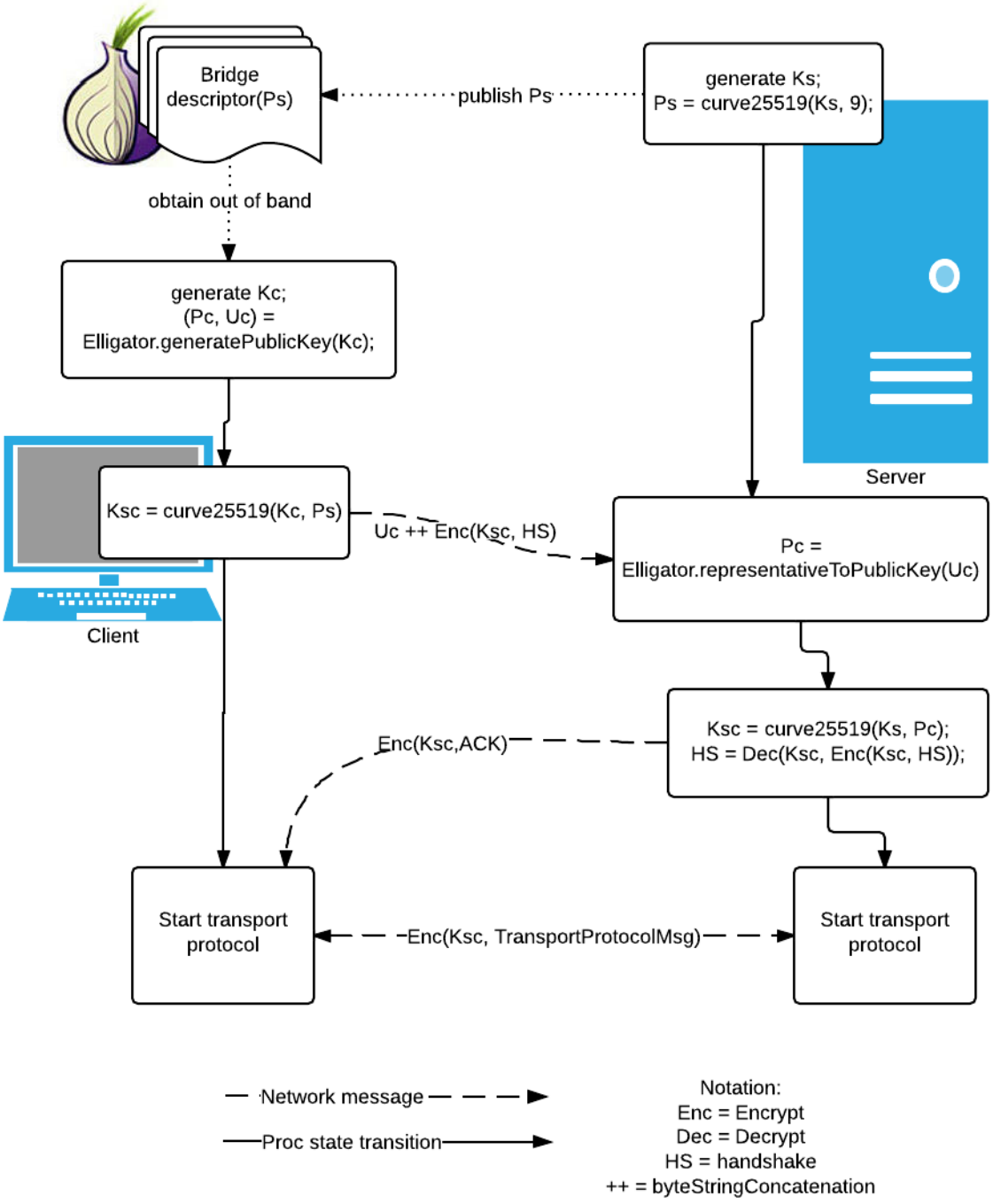

Figure 5.2: Cryptographic protocol

This mode of operation has the following desirable properties:

- provides data **confidentiality** - contents of the message are kept secret
- provides data **integrity** - the decryption function of AES in GCM mode has a return value which is either a plaintext message or a failure to decrypt. The function fails to decrypt whenever the input data hasn't been correctly encrypted (it either has been tampered with or it simply isn't an encrypted text)

The first property works to achieve our confidentiality security goal.

The second property is necessary to determine whether an encrypted payload is actually *present*. From the perspective of the receiver an incoming BitTorrent piece may or may not contain an encrypted payload. Decryption is attempted on the content of all BitTorrent piece messages arriving: if it fails it is considered to be an ordinary piece, if it succeeds then it had an encrypted payload.

5.4 Transport protocol

describe the transport protocol - this is the protocol of the plugabble transport embedded in the BitTorrent traffic; it talks about handshakes, acknowledgements, data messages, switch swarm messages etc.

5.5 Steganography

BitSmuggler puts the transported traffic in the Piece messages of a BitTorrent connection. Through the use of steganography, the transported traffic is concealed in the data file that the BitTorrent connection is exchanging so that the adversary is not aware of the presence of a secret message.

The steganographic problem at hand is different from the classical applications of steganography: the file data which contains the concealed message will not be inspected by people (eg. looking at a picture of a cat which contains a hidden message), but only by automated steganalysis verifications.

The challenge here is to make it good enough so that any kind of real-time steganalysis performed by the adversary is not able to detect the presence of a concealed payload.

We employ a *simple technique*: place payloads encrypted with **symmetric key encryption** using **AES-GCM** as the block cipher and its mode of operation in the content of files whose data looks like random noise. When we say a piece of data looks like random noise, we mean it is a sequence of bytes that has a probability distribution close to that of random noise.

The result of AES-GCM encryption is a sequence of bytes whose probability distribution is that of random noise and therefore it is indistinguishable from a random string if one does not know the secret key. However it is distinguishable from a random string if the secret key is known. Thus it should blend in with the content of the file.

As far as the initial handshake message is concerned, the choice to use Elligator ensures that sending the server public key in plaintext does not compromise the steganography by making it look random and therefore blend in as the rest of the cyphertext messages.

Here we make a **key security assumption**: the compressed media traffic is not (easily) distinguishable from a random bit string.

This simple steganographic technique relies on advantageous properties of BitTorrent protocol/traffic:

- **files sent in pieces** - any analysis of the file as a whole requires reassembly of the pieces being sent - which is extremely hard to do in real-time for all BitTorrent traffic passing through the national network perimeter
- **videos are the most popular torrents** - video files have good steganographic properties for our purpose. The more video torrents there are, the easier it is to blend

Video file steganography

Video is the preffered type of file for embedding concealed data in BitSmuggler . This preference is based on the following properties:

- video file content is compressed - codecs compress video streams creating data with high entropy (compression eliminates redundancy) which looks close to random data and therefore looks like the data encrypted with a symmetric key
- video files are large - allow for a longer exchange of data using the same file. No need to change the file/swarm quickly

5.6 Active probing resistance

Censors use a practice called active probing to check if a remote sever is indeed a Tor bridge. They simply try to contact it pretending they are Tor clients, or, even simpler, they send junk bytes at it to see if responds with the Tor protocol. There is empirical proof of this strategy being used by The Golden Shield project ([Wilde, 2011]).

BitSmuggler makes it impossible for an adversary to perform active probing on the Tor bridge without knowing the Tor bridge's descriptor. If an adversary attempts to make a BitTorrent connection and thus initiate a BitSmuggler connection, the BitSmuggler

server will only regard it as a valid connection and respond accordingly if the initial message contains proof of knowledge of the server's public key by being correctly encrypted as described in the cryptography section (5.3). If it does not receive this message, it lets the BitTorrent connection flow untampered and classifies it as a non-BitSmuggler connection. To the adversary, it seems that the remote BitTorrent client responds normally - performing a standard peer connection.

If the bridge descriptor is known by the adversary, active probing is possible since the adversary is indistinguishable from an honest user, as far as BitSmuggler is concerned.

This behaviour is implicit in the design: since it is using a normal BitTorrent client to generate the BitTorrent traffic it responds as a normal torrent client would when contacted by peers. It just check all incoming connections to see if they are BitSmuggler connections and *only then* does it start embedding encrypted responses in the BitTorrent pieces.

Chapter 6

Implementation

The goal of the system is to pipe traffic from the Tor client to the Tor bridge and back through the use of the bittorrent traffic as a camouflaging transport channel whick masks the existence of a Tor connection.

The current system implementation provides a reliable communication channel camouflaged as BitTorrent traffic between the client and the server of any client-server model application. This means that even though the current use case of the system is to act as a pluggable transport for the Tor network, channeling traffic between a Tor client and a Tor bridge, it is general enough to be used as a communication channel between any client and server.

The implementation consists of a pair of pluggable transport (PT) client-server processes, each accompanied by a BitTorrent client process under its control, both running alongside their user client and server, respectively. From this point onward we shall refer to the client and server *using* the PT processes to communicate as *user client/server* and their traffic will be called *user traffic*.

The server and the client communicate through a custom *transport protocol* whose main purpose is to proxy user traffic back and forth.

The PT client-server processes are implemented in Haskell, a choice justified by the language's powerful type system which helps with achieving correctness and code reuse, and its very good libraries for data streaming and concurrency. Work on the project resulted in the creation of many useful reusable packages that, once refactored accordingly, can hopefully be added to the Haskell official package repository.

6.1 System architecture

The system architecture follows a client-server model. The client and the server use a separate BitTorrent client process each to generate authentic Bittorrent traffic between each other. Once a peer-to-peer connection is established between the 2 torrent clients the BitTorrent traffic flow starts transporting the user traffic back and forth. The technique used to make the BitTorrent traffic "carry" the user traffic is to replace the payload of *some* of the BitTorrent piece messages with transport protocol traffic.

Figure 6.1 shows the components of the server and the client for the *concrete* use case of the system to create a communication channel between a Tor client and a Tor bridge. These components are the processes that the client and the server run and how connections are made between them.

The client has the following components:

- Tor client process which generates the user traffic that needs to reach the the Tor bridge
- Pluggable transport client process that accepts connections from the Tor client process and proxies its traffic
- Socks proxy facing the Tor client as an interface for proxying the Tor traffic (runs as part of the pluggable transport client)
- BitTorrent client which initiates a peer connection to to the server-side bittorrent client in order to exchange file pieces of a file advertised by the Tor bridge off-band
- Socks proxy facing the BitTorrent client which proxies its traffic in order to allow the PT process to read incoming transport protocol messages and embedd outgoing transport protocol messages (runs as part of the PT client)

The server has the following components:

- BitTorrent client waiting to receive peer connections that will "carry" the transport protocol traffic;
- Reverse proxy for the incoming BitTorrent connections to the BitTorrent client living on the server-side (runs as part of the PT server process)
- Pluggable transport server process that watches all incoming Bittorrent connections hitting the reverse proxy and selects the Tor client ones to proxy further to the Tor bridge
- Tor bridge which receives incoming connections from Tor clients in order to relay their traffic through the Tor network; in this setting the traffic from its clients is proxied by the PT server process process which opens connection the the Tor bridge using the ORPort protocol

The distinction is made between the 3 types of traffic:

- Tor traffic (blue) is generated by the Tor client and Tor bridge and flows back forth to the plugabble transport process which wraps it in the transport protocol traffic
- Transport protocol traffic (purple) is generated by the plugabble transport process and embedded in the BitTorrent traffic
- BitTorrent traffic (green) is flowing over the network and through the adversary firewall between the the 2 bittorrent clients while being subject to inspection and tampering by the Socks and reverse proxies in order to read and embed the incoming and respectively outgoing transport protocol traffic

6.1.1 The life of a Tor data byte

In order to better understand how the whole data channel works we will describe step by step, the lifetime of a data byte after it is sent by the Tor client and until it reaches the Tor bridge. This is described assuming the whole connection has already been established and intiation/handshaking completed. Note that for all other purposes the data is reasoned about as a stream, but we look at a single byte to understand the flow.

A byte is sent by the Tor client process to the Tor bridge. It is proxied to a localhost socks proxy that the Tor client is configured to use. It is then picked up by the PT client process who controls the socks proxy. The PT process then waits for a BitTorrent piece to take the byte over the network. The PT process runs another socks proxy facing the Bittorrent client. It can thus see all incoming and outgoing BitTorrent messages going through a connection *initiated* by its BitTorrent client. When a piece message created by the *local* BitTorrent client is seen by the PT process, it tampers with its contents, removes its old content (a part of the file it was transporting) and inserts the byte encrypted.

The BitTorrent message travels across the network through the adversary's firewall (hopefully unseen, unheard) and reaches the server machine. It is then proxied through the reverse proxy standing in front of the BitTorrent client living on the server side. This reverse proxy is owned by the PT server process which looks at all incoming piece messages to see if they are holding transport protocol payload. When the travelling byte's message reaches it, it decrypts the whole payload, extracts the byte, sends the piece further on to the server-side BitTorrent client. It then sends the byte through the open Extended ORPort connection to the Tor bridge process running locally.

In a similar, but not identical way (since there is assimetry between the client and server) a byte travels from the Tor bridge to the Tor client.

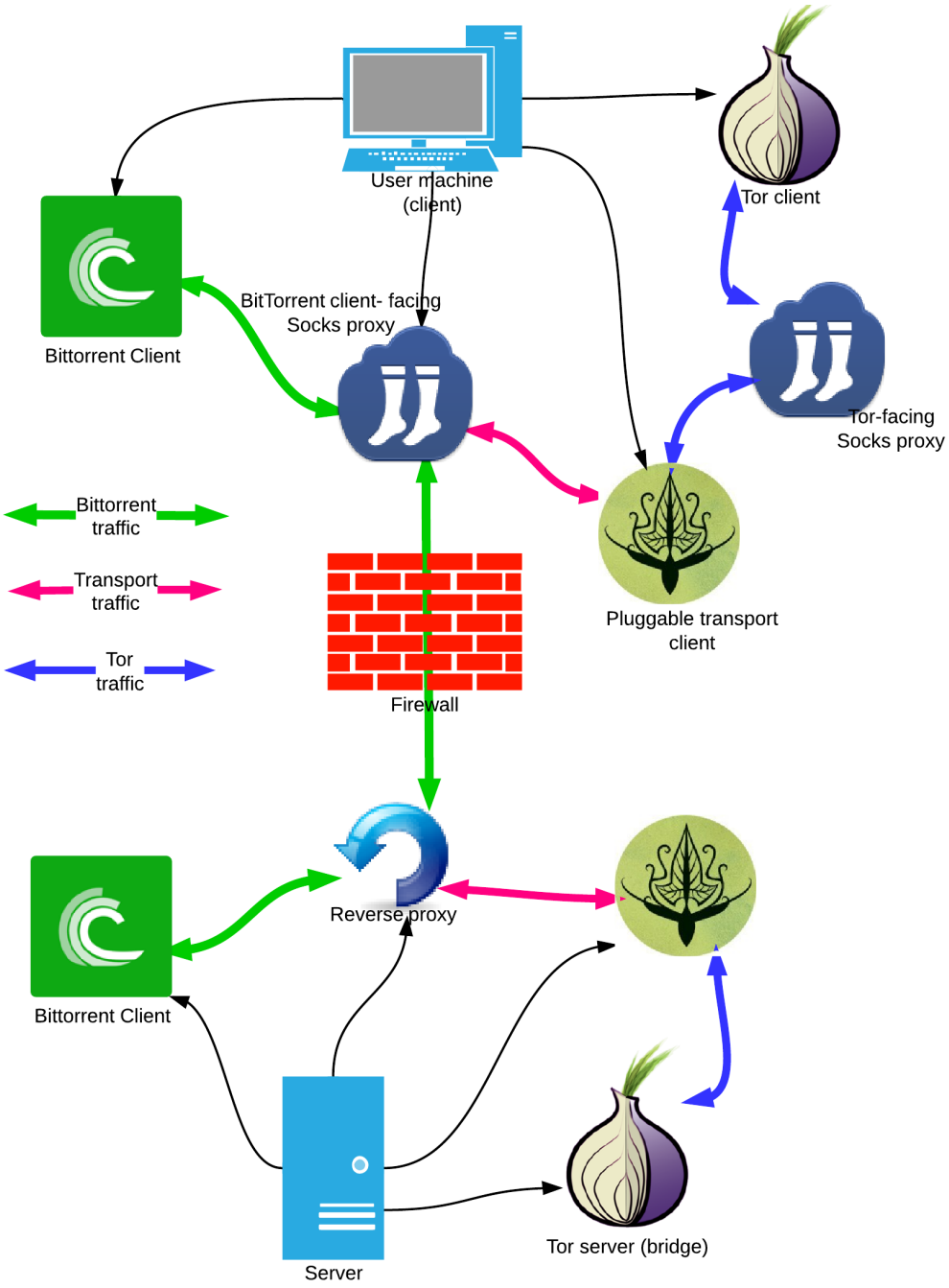

Figure 6.1: System overview

6.1.2 BitTorrent client interaction

A critical design choice of this project is that to use a *real* BitTorrent client to generate BitTorrent traffic between the client and the server. This choice was made in order to obtain an authentic BitTorrent traffic pattern that is indistinguishable from genuine traffic. However, this came with the tradeoff of a more complex implementation.

As we saw above, using a BitTorrent client to generate transport traffic involves proxying its connection with a socks proxy on the client (because it's an outgoing connection) and a reverse proxy on the server-side (because it is an incoming connection). Furthermore, in order to correctly read and tamper with the BitTorrent messages, the entire stream of the BitTorrent connection is parsed and then serialized back on its way out of the proxy.

The BitTorrent client in use needs to be controlled by the PT process. Therefore, the client needs to support the following actions in an automated fashion (i.e. exposing a network API/command line interface):

- socks proxy configuration
- Adding a torrenfile/magnet link
- control per torrent file pausing, resuming, upload/download speed, adding peers

To fully achieve the goal of creating indistinguishable BitTorrent traffic, the sum of the usage share of the BitTorrent clients employed by the system needs to be high. To undestand this, think about the extreme situation in which the system uses only one particular obscure BitTorrent client with a 0.1% market share, in which case the adversary can focus his detection efforts on the connections which involve only those particular clients, as deduced from the plaintext handshake of the BitTorrent protocol.

Given the requirement above, the system is designed to support multiple types of BitTorrent clients which in turn are required to support the features mentioned above to allow for control and proxying. For this purpose, a general BitTorrent client interface was designed, which requires implementations for each client. The current codebase contains control modules for the UTorrent client and the Deluge client. The UTorrent client was chosen since it comes up as being either the first or second most popular client based on information collected by trackers, while the Deluge client was selected because its deluged daemon is ver configurable allowing for fine-grained control, making it the best fit for initial prototyping and demos.

The following snippet shows the Haskell data type associated with a BitTorrent client connection:

```
data TorrentClientConn
  = TorrentClientConn {
      addMagnetLink :: (MonadTorrentClient m) => String -> m (),
```

```
addTorrentFile :: (MonadTorrentClient m) => String -> DB.ByteString -> m
    TorrentHash,
listTorrents :: (MonadTorrentClient m) => m [Torrent],
pauseTorrent :: (MonadTorrentClient m) => TorrentHash -> m (),
setProxySettings :: (MonadTorrentClient m) => [ProxySetting] -> m (),
connectPeer :: (MonadTorrentClient m) => TorrentHash -> HostName ->
    PortNumber -> m ()
```

6.2 Tools

add ian intro to the tools sections

6.2.1 Programming language

When choosing the programming language for the project the main criteria were:

- is high-level
- favors code reuse
- favors correctness and security
- has reliable network, concurrency, cryptography libraries
- the language toolchain (compiler/interpreter, profiler) allow you to produce efficient code in terms running time and space performance
- is a memory-safe language - strongly suggested feature by the Tor project developers ([Project, 2011c])

Haskell is the programming language of choice for this project. Haskell is a high-level purely functional language that is advertised as being good for the rapid development of correct, robust software. It's type system is both flexible and powerful allowing you to write code that is concise and reusable while at the same time subjected to a great deal of compile time checks.

Atlhough not a requirement, the fact that the language allows you to write pure computations (no side-effects, equivalent of mathematical functions) combined with the strong compile time checks makes it much easier to write correct code. Whenever it is the case below, I will emphasize when a particular code component has been implemented in a pure fashion to facilitate testability and achieving correctness.

The fact that the Haskell toolchain allows for writing of performant network applications has been proved empirically by the creation network applications such as the Warp

HTTP web server whose performance exceeds that of other modern systems (node, tornado, php) [Snoyberg, 2011].

As for the availability of libraries, the choice of programming language was not the correct one in terms of prototyping and quick results. I believe Python would have been the obvious correct choice. If I were to choose Python the following libraries/modules would have been available:

- Twisted networking engine
- socks/reverse proxy server
- Extended OrPort protocol
- BitTorrent protocol parser
- REncode
- deluged RPC API
- uTorrent RPC API

The parsing, encoding and protocol modules above needed equivalent implementations in Haskell. Haskell doesn't have a direct equivalent of Twisted, namely a framework for writing custom network applications. However, one can argue that by combining existing Haskell libraries you can get equivalent functionality with not much extra boilerplate code, as we shall see in the libraries section.

Not only this but Python, being high-level, dinamically typed, and generally well designed, is a very good choice for prototyping and writing programs whose final shape is not known and where signficant design changes happen while implementing. It can be argued that this was the case for this system.

Two other aspects which greatly affected productivity but were not amongst the main considerations when choosing the programming language are *familiarity* and *learning curve*. I was moderately familiar with Python and it has a short learning curve, while I was a beginner Haskell programmer and the language has a significantly longer learning curve. Thus, a lot of the time spent on the project was invested in learning Haskell.

Having said that Python would have been the correct choice for prototyping, I would argue that Haskell is the long term correct choice, since it meets the other stated criteria and the minor lack of libraries is something that can be compensated by just investing development effort.

6.2.2 More on Haskell and correctness

Correctness in crucial in security systems since one single bug can compromise the security gurantees of the implemented solution.

Haskell emphasizes safety more than other statically-typed languages and allows for more complex static checks that can go as far encoding static proofs using the Haskell type-system which prove parts of your program correct at compile-time (see work by Oleg Kyseliov [Kyseliov, 2004]).

Often sofware security breaches rely on triggering unexpected IO actions in the target program. By using Haskell's typeclasses we can restrict what IO actions can be performed in a certain computation.

For example, if a monadic computation of type m is supposed to load configuration files from a particular directory in the filesystem, instead of qualifying it as a (MonadIO m) you can qualify it at MonadDirReader which is constrained through the interface that MonadDirReader provides to only perform certain IO actions, reducing the possiblity of erroneus IO actions to only 1 place: the iplementation for MonadDirReader.

Having said that, the current prototype of BitSmuggler does not make use of these more advanced features (given it is a prototype) but its future iterations may incorporate them.

6.2.3 Libraries

On the flip side, the Haskell libraries that are available are reliable and efficient and superior to libraries in other languages solving similar problems.

Based on importance in the project, the following libraries are worth mentioning:

- Control.Concurrent - offers explicit concurrency abstractions.
- Control.Concurrent.STM - software transactional memory implementation
- Data.Conduit - library for handling streams of data
- Data.Attoparsec - library for writing parsers in a concise, declarative way

Control.Concurrent

I used this library for concurrent handling of requests and tasks.

The main primitive of the library is forkIO which allows you to spawn a thread that is sent off to perform a computation. As of GHC 6.12, that thread is not an OS level thread, but a user-space green thread, which means many (read: millions) of such threads can be created without being concerned with expensive creation, per thread memory consumption, context switching and finalization. These threads run in what is called an event loop.

The nice thing about the Haskell implementation of green threads is that for the user (programmer) the IO actions of green threads are still expressed naturally in the IO

monad with no changes required. Other environments supporting green threads such as NodeJS, Dart or Python Twisted force you to write code in a continuation-passing-style with a callback or defferred object for each IO action.

The following code snippet illustrates the loop handling new connections in the proxy module. Each new connection is handled independently by the handleConn function.

```
loop config serverSock
  = forever $ accept serverSock >>= liftIO . forkIO . (handleConn config)
```

Nonetheless, the underyling GHC implementation for threads is using an even-driven model which makes efficient use of all machine cores, all being abstracted away from the user (the programmer). It can be said that Haskell makes the best of both worlds (the event driven concurrency and threaded concurrency model).

Control.Concurrent.STM

This library provides safe access of shared memory where all accesses are modelled as transactions (hence the name).

My main use case for this library are message channels. In order for the threads to do useful work together, they communicate by sending messages to each other through channels. Although implemented in the STM library, channel communication should be understood as "no-shared-memory" way of communicating where a thread pushes a message inside a channel and another thread reads it from that channel.

The next code snippet shows how the socks proxy thread notifies the PT client thread that it has finished initializing the proxying of a BitTorrent connection by sending a message through a control channel.

```
-- proxy initialization function
liftIO $ atomically $ writeTChan control Done
-- ...
-- the PT client thread
 completion <- liftIO $ timeout btConnInitTimeout
                                (liftIO . atomically . readTChan $ control)
 when (isNothing completion) $ throwError "bittorrent connection start timeout"
```

Data.Conduit

The conduit library is used for modular and efficient handling of data streams. I used it for handling the data streams of the proxies and for processing the stream of incoming and outgoing transport protocol messages in a pipeline-like fashion.

The next snippet shows the how the stream of incoming BitTorrent Piece payloads is handled in order to be transformed in transport protocol messages:

```
streamIncoming msgChan payloadChan decrypt =
   (chanSource payloadChan)
   $= Conduit.map decrypt --attempt to decrypt all incoming payload
   $= filter (not . isNothing) $= Conduit.map fromJust -- filter the
       succesfully decrypted ones
   $= conduitParser parseTransportMessage -- parse the messages
   $$ chanSink msgChan -- send them down the message channel
```

Notice how this style of coding isolates the reading and writing of data IO actions (in this case reading/writing channels) from the logic of the transformations on the stream which are pure computations that can be tested separately.

I also used the Data.Conduit library for a more unconventional purpose: writing network protocols. A network protocol was modelled as a conduit that takes as an input a stream of bytes and produces a stream a bytes or a final result. The next snippet illustrates the data types and the function used to run the protocol:

```
type ProtocolConduit a = (MonadNetworkProtocol m) =>
                                      Conduit ByteString m (ProtocolOutput a)
data ProtocolOutput a = ProtoData ByteString | FinalResult a
runProtocol :: (MonadNetworkProtocol m, MonadIO m) =>
                       (ProtocolConduit a) -> Socket -> m (Maybe a)
runProtocol protocol sock
  = (sourceSocket sock) =$ protocol $$ (sinkSocketWithResult sock)
```

Thus, to implement the the ExtendedOR Port initialization protocol, one needs to implement a Conduit computation with the ProtocolConduit signature.

This achieves the following: the protocol logic is implemented as a *pure computation* in the form of a stream transformation decoupled from the IO actions of reading and writing to network sockets.

Data.Attoparsec

Attoparsec was the parser library of choice used for parsing the message streams of all the implemented protocols because the library was build with this particular task in mind.

The library allows for expressing parsing functions in a concise, pure manner as shown in the following snippet for parsing the socks5 handshake message:

```
parseHandshake :: Parser Handshake
parseHandshake = do
  versionByte <- anyChar
```

```
  methodCount <- fmap ord anyChar
  methods <- replicateM methodCount anyChar
  return $ Handshake versionByte methods
```

6.3 Code structure

The codebase was developed with a focus on building reusable modules. At this stage the code requires more refactoring work, documentation and optimization to meet Hackage standards, but as it stands it is clearly structured and abstract enough to be reusable.

The user facing Haskell modules are the Client and Server modules. The next snippets give a feel about how to use these modules for communicating through a BitTorrent camouflaged channel (no Tor integration just yet):

```
-- Client interface

   transport <- Client.init socks5ProxyPort torrentClientConnection
   let info = serverInfo clientEncyption serverTorrentFile serverSockAddr
   conn <- Client.makeConn transport info
   connectionPut conn "why hello there"
   response <- connectionGetChunk conn
```

```
-- Server interface

{-
receiverPublicPort - the publicly advertised BitTorrent port for incoming
    connections
receiverPrivatePort - the port at which the BitTorrent reverse proxy listens
    and where all BT traffic is redirected on
-}
Server.run echoHandleConn receiverPublicPort
                                      receiverPrivatePort
                                      serverEncryption
                                      torrentClientConnection
echoHandleConn conn
  = forever $ (connectionGetChunk conn) >>= (connectionPut conn)
```

The rest of the modules are auxilliary modules but many are standalone pieces of code that can be integrated as individual packages in Hackage.

- NetworkProtocol - module defining an interface for writing and running network protocols as Conduits
- Proxy - customizable proxy with a hook in connection initialization and Conduit hooks into the incoming/outgoing traffic; has logic for running socks proxies and

reverse proxies

- Extended OrPort protocol - Tor protocol for connecting a pluggable transport to the Tor bridge
- BitTorrent Message - parser and serializer for the stream of BitTorrent messages. implemented using Attoparsec and the Data.Binary package
- BitTorrentClient - module defining a general interface for BitTorrent client communication
- deluged RPC API - library for communicating with the deluged daemon from Haskell
- uTorrent Web API - library for communicating with the uTorrent server from Haskell
- REncode - library for an encoding algorithm of loosely structured data. It is the Deluge project's more optimized version of the original BEncode encoding created for the BitTorrent project
- Stream - contains data stream handling functionality common between the Server and the Client
- Utils - module containing miscellaneous useful functions

mention things unimplemented yet: the crypto, swarm changing

6.4 Tor integration

The system runs as an externally managed Tor pluggable transport and doesn't make use of any of the existing Tor modules/frameworks for plugabble transports.

Many of the other Tor pluggable transports (StegoTorus, ScrambleSuit, Dust) make use of the Tor obfsproxy framework for developing pluggable transports. For developing this particular pluggable transport the obfsproxy framework isn't flexible enough. The reason is that obfsproxy handles the connection to the Tor server and client itself allowing you to "plug in" the transport logic in the middle, while in the case of our system there is no direct connection between the Tor client and bridge. The only connection over the network is the Bittorrent connection.

As stated previously, the choice of programming language prevented any code reuse since obfsproxy and most Tor plugabble transports are developed in Python, C++ or more recently Go(meek). However, by developing the current solution we set foundation for the development of Tor pluggable transports in Haskell: the custom socks server module and the ExtendedORPort connection are the ncessary glue to handle Tor client and Tor bridge connections, respectively.

6.5 Missing implementation pieces

Due to time constraints, not all implementation goals were achieved. The following components/features of the project are missing an implementation.

Encryption

Encryption implementation is missing and it currently has a dummy implementation. The code is designed to allow for encryption scheme changes based on changing just 1 argument in the initialization function and the interface to the encryption data type is flexible enough to allow for different types of encryption. The reasons for not completing this are that Haskell doesn't have an implementation or bindings for an implementation of Elligator. At the same time, there is no official Hackage package for bindings to a curve25519 library and the existing bindings are still under development.

Swarm change

BitTorrent swarm change has not yet been implemented. The cookie mechanism is not yet implemented either. This is mostly because a proof-of-concept implementation could work without this feature.

Bridge handling many clients

Efficient handling of many clients connecting to a Tor bridge has not been implemented. In order for the connection throughput to stay at a reasonably good level a scheme where clients are load balanced over a set of swarms needs to be developed. This task is a significant undertaking by itself.

Chapter 7

Evaluation

The evaluation of BitSmuggler consists of:

- assessing whether the security goals of the project presented in section 3 are achieved by the design
- a performance measurement of the system.

It must be noted that, since BitSmuggler develops a completely new technique in terms of "cover traffic", no attack exists against it at the time of evaluation. Thus, there is no relevant benchmark of its efficiency in terms of security. Therefore, the evaluation of how the project achieves its security goals is *theoretical*.

A concious time management decision was made to not attempt to design potential attacks against the system as a form of evaluation and opt for a theoretical evaluation only, instead, supported by statistical analysis arguments.

Stegotorus [Weinberg, 2012]took the approach of designing attacks against their own design and evaluate their project's security in terms of how well it performs against those attacks. Their circumstances were different though: they were a team of 7 security experts from Standford so apart from having more manpower they also had the advantage of more minds looking at the design from different angles - which is a crucial aspect in the review of security designs. In spite of this, the attacks devised in "The Parrot is dead" [Houmansadr, Brubaker, and Shmatikov, 2013] were found to be effective against StegoTorus. This proves the point that it is good to look into potential attacks as a separate problem, after the project has been put out in the open for peer reviews.

7.1 Theoretical evaluation

The theoretical evaluation is done in the form of arguments for and against the design choices given the stated security goals.

7.1.1 Preservation of Tor security goals

The Tor unlinkability goal is maintained since BitSmuggler does not interfere with the Tor protocol. It just acts as a transport channel for Tor traffic to the target Tor bridge, which is the first node in the Tor circuit.

Even in the case in which the undetectability and the confidentiality goal (at the BitSmuggler encryption layer) are violated the only thing that is revealed is the encrypted traffic of a client to a Tor bridge. In this case, it is as though the client communicates to a normal Tor relay.

The performance goal (not a security goal) is shown to be preserved in the performance evaluation section.

7.1.2 Cover traffic choice evaluation

The choice of cover traffic was made to achieve the **undetectability**, **unblockability** and **performance** goals.

BitTorrent traffic was shown to be the most prevalent type of internet traffic in Asia, the second most prevalent in Europe and the fourth most prevalent in North America. This means that an adversary aiming to detect BitSmuggler traffic needs to monitor a huge amount of traffic making detection a hard problem.

It also means that blocking it completely would affect a large number of users and would remove the main channel of distributing user data in many countries. Based on this, banning the BitTorrent protocol is considered an extreme choice which can have a serious socio-economic impact.

While an adversary may be willing to ban encrypted BitTorrent traffic in the way some real-world adversaries blocked ssh, https and vpn traffic, it is much more unlikely for the plain-text version of BitTorrent to be blocked. The current design works with the plain-text version of BitTorrent.

Choosing BitTorrent as the cover traffic allows for performance in terms of high throughput. BitTorrent clients download and upload large amounts of data so they create a lo of traffic which ca implictly cover a lot of transported data.

In the grand scheme of anti-censorship protocols that use existing protocols as cover protocols, BitTorrent was the big name on the list of unexplored cover protocols. Its properties show great potential.

7.1.3 Cryptographic protocol evaluation

Cryptography in BitSmuggler addresses the security goals of **confidentiality** and is designed to be used with steganography to address other goals as described in the steganography evaluation section 7.1.4.

All messages sent between the client and the server are encrypted through the use symmetric key encryption with a shared secret obtained through a Diffie-Hellman key exchange. Even in the case in which the adversary is able to violate the undetectability security goal, the contents of the message exchange are kept secret.

The only thing sent in plain-text is the public key of the client to the server. As in any Diffie-Hellman key exchange, if the adversary learns about it the confidentiality of the message exchanged is preserved.

Protocol assumptions

The cryptographic protocol makes a strong assumption about the fact that the client obtains the server's public key out-of-band and that is Tor bridge descriptor is trusted to be correct. The issue of Bridge descriptor acquisition is considered out of the scope of this project and is delegated to the Tor mechanisms of Bridge descriptor distribution.

Making this assumption means it is necessary that the current Tor descriptor format is augmented so that it contains the extra information (public key, bootstrap torrent file) required by BitSmuggler . It also means that for any other type of communication system that uses BitSmuggler as a cover channel, a different way of distrbuting the server descriptor needs to be found.

The assumptions are considered reasonable.

Tor bridge descriptors can contain a small piece of extra information. Other pluggable transport protocols require an extra piece of information as well for similar purposes (Dust, SkypeMorph).

Trusting the correctness of the Tor descsriptor is reasonable. Tor bridge descriptor distribution is a separate problem, and the current project focuses on the particular problem creating a cover channel for Tor traffic.

Cryptographic primitives choice

The choice of using curve25519 as the function for ECDH is good based on the fact that it is not developed by NIST (eg. curve P-256) but by Daniel J. Bernstein. NIST is under the influence of NSA, who was found to be introducing backdoors in their promoted cryptographic tools [NICOLE PERLROTH and SHANE, 2013].

Choosing curve25519 offers advantages in terms of performance [Bernstein, 2006], but those are not particularly relevant. The only place where the choice of Diffie-Hellman function matters in terms of perfomance is the server's ability to handle many connections. In our case is a non-issue since it is negligible compared to other performance bottlenecks (eg. handling many BitTorrent connections).

Elligator is designed with anti-cesorship protocols in mind and the goal of undetectability. It reduces protocol complexity [**?**] for every protocol which wants to achieve indistinguishability. The only temporary drawback is that, at the time of writing, there is no audited implementation of Elligator.

Protocol complexity

Compared to other pluggable transports' cryptographic protocols, the BitSmuggler protocol is arguably simpler, while achieving the same security goals.

The more interesting part of the protocol is the handshake, which relies on sending the public key of the client in plain but concealed to look like a random string by using Elligator along with the handshake message encrypted with the computed shared secret. After the handshake step, it uses a standard shared secret encryption performed with AES in GCM mode.

By contrast, ScrambleSuit ([Winter, 2013b]) uses session tickets to allow reconnections. This complicates the cryptographic protocol but by doing this it makes it easier for the server to keep track of sessions. Their alternative authentication protocol is more complex as well, making use of an extended uniform Diffie-Hellman scheme to defend against active probing.

Also, StegoTorus [Weinberg, 2012] uses asymmetric key encryption which needs additional machinery to make its ciphertext indestinguishable from random strings since it doesn't have have this property by default. Thus, they use Möller's encapsulation mechanism to achieve this which creates further complications in the design. Note that this design choice was made before Elligator was created so

Simplicity is a big plus for BitSmuggler since the more complicated a cryptographic protocol is, the higher the probability that it is flawed.

7.1.4 Steganography evaluation

Steganography in BitSmuggler works to achieve the goals of **undetectability** and **plausible deniability**.

A key property of the design of BitSmuggler is that it does not create any *fingerprintable string patterns* in the traffic that it generates. This is ensured by the following design decisions:

Data	Mean	Standard deviation
Video files	7.991141	0.01074686
Encrypted data	7.999814	1.888357e-05

Table 7.1: Comparison of entropy values

- preserving the BitTorrent stream structure - only the Piece payloads of the BitTorrent stream are altered and they are the only messages of the protocol which contain custom data
- use of the Elligator encoding - using it to send the client public key ensures that no message contains a plain-text key whose mathematical properties can be verified - the key looks like a random string
- use of AES-GCM - the block-cipher mode choice for the symmetric-key encryption perfomed by the protocol ensures that the encrypted payloads looks like random data

The steganographic design of BitSmuggler relies on a important assumption stated in section 5.5, namely the fact that the inserted encrypted payloads are indistinguishable from the media file data they replace.

In order to evaluate this claim, we make a statistical argument about the properties of encrypted payloads and video files and how they are indistinguishable based on their entropy, using **Shannon entropy** as a measure. The Shannon entropy is a measure of information content and since both types of data have very little redundancy, they both exhibit an almost maximal entropy value.

Shannon entropy is measured for byte sequences of length 1. Based on the computational power assumptions made in the threat model, it is very unlikely that a Shannon entropy on longer sequences can be measured given the hard bound to computation time.

We measured the Shannon entropy for a set of video files and for a set of sequences of encrypted data. The results are shown below.

Although both results are close to a maximal value (with a 256 symbol set, maximum entropy value is $\log_2 256 = 8$) and both have a small standard deviation, a difference can be spotted.

We used a **Kolmogorov–Smirnov statistical test** which compares the 2 samples and evaluates the hypothesis that they both originate from the same distribution. Based on the 2 sets of samples , the test shows that the hypothesis is unlikely. It yields a p-value of p-value = 5.64e-05. Based on the p-value we decide on whether to reject the hypothesis. Standardly, the hypothesis is rejected for values less than 0.5 so based on the results of the test we *reject* the hypothesis.

The above tests shows that 2 types of data can be differentiated by an adversary if he is able to recompose the whole file content.

Recomposing the whole file content involves:

- eavesdropping on the BitTorrent connection
- reconstructing TCP packets
- extractring the Piece message payloads from the BitTorrent communication stream

We stated in the adversary model that all these actions are not computationally feasible so therefore an adversary will not be able to perform statistical test such as the one described above.

In a realistic scenario, the adversary performs a statistical test on incoming TCP fragments. This means that the pure file data is mixed with protocol data. It is true that the bulk of the data flowing through consists of the file data, so the above test may still be effective even under these conditions, but no verification of this possibility has been made yet.

It must be noted that the current implementation did not use strong steganographic techniques: the transported encrypted data is not embedded in a video file by preserving headers and subsituting parts of the data. It just completely displaces the video file data in the BitTorrent piece in which it is inserted. Adding this feature is a reasonable undertaking and its single side-effect will be an increase in steganographic expansion (trading performance for security).

Interpretation

Based on this evaluation, we conclude that the design which leaves no fingerprintable patterns is sound and the only practical way of cracking the steganographic technique is through statistical analysis.

Furthermore, the approach of displacing video data and replace it with transported data shows great promise in the sense that the video data's entropy is very close to that of a random sequence while the adversary's computational capabilities of detecting the difference are limited.

However, we believe, based on the statistical analysis of encrypted data versus video data, that there is a possibility that a fast implementation of some entropy measure, even if it operates on TCP fragments, may prove effective in violating the undetectability goal. As stated, the current steganography approach is a crude one, a stronger steganographical approach should be investigated which is very likely to alleviate this shortcoming.

Wiley's Dust project [Wiley, 2013] has a solution for making encrypted data match a desired statistical distribution which is *exactly* the thing that BitSmuggler needs to overcome the shortcoming discussed above. Conveniently, his project is developed in Haskell so integration should be smooth.

As far as plausible deniability is concerned, since no fingeprintable trace is present, all data is encrypted and the cyphertext looks like a random string there is no proof that

the BitTorrent connection was not authentic and that encrypted data is present. This argument can be made if the acusing side does not have access to the user's machine.

7.1.5 Piece hash attack

Since the payload of the piece messages of has been tampered with, as a tampered piece travels between the BitSmuggler client and server it may be checked for consistency by an adversary.

We deem such as an attack *computationally infeasible* based on our adversary model: extracting piece payloads involves the same actions of reconstructing the BitTorrent messages as those mentioned in the steganography evaluation section. Furthermore, to get the hash of the piece, the adversary also requires a copy of the torrent descriptor file which is even more computational work.

As an alternative to eavesdropping, an adversary may join BitTorrent swarms to check if any of the swarm peers is consistently sending tampered pieces and therefore suspect its authenticity. This is impossible thanks to the implementation decision off ensuring that before any communication is started, the 2 BitTorrent clients already have a correct copy of the file which will be used as "carry traffic". The BitTorrent client software itself only operates with pieces which pass integrity checks, since each time a tampered piece arrives, the BitSmuggler proxy reads it contents and corrects the piece contents by reinserting the valid piece payload.

7.1.6 Packet arrival time attack

Even though BitSmuggler changes the BitTorrent message stream in ways that preserve undetectability based on packet content, it may be changing in a subtle way the pattern of packet inter-arrival times.

This pattern change can be caused by the fact that it proxies the stream of BitTorrent traffic and performs computations of varying durations on the stream before forwarding it to the destination. If it creates a specific pattern of packet inter-arrival times that is statistically *distiguishable* from that of normal BitTorrent traffic, than an attack relying on the observing this pattern is considered *feasible* as previous work has shown ([Mohamad Jaber and Barakat, 2011]).

According to our model of the adversary such a statistical analysis is possible. Therefore, the design should consider this potential attack and at the time of writing it does not.

By comparison, ScrambleSuit [Winter, 2013b] addresses this issue by controlling its packet inter-arrival time. There is nothing in the design of BitSmuggler that prevents it from implementing a similar scheme, so this a problem that needs to addressed in future work.

7.2 Performance evaluation

The performance evaluation is meant to show how efficient BitSmuggler is in transporting data in terms of time and resources (camouflage traffic). BitSmuggler was evaluated stand-alone, without the Tor integration, running a minimal custom protocol for sending files.

7.2.1 Testing setup

The setup for all performed experiments consisted of:

- a client that was running on a laptop PC with a wireless connection (6.20 Mbps download. 085 Mbps upload).
- a server running on a virtual host in Amsterdam.

The round trip time between the 2 machines is shown in table 7.2:

Mean	64.698 ms
Standard deviation	38.830 ms
Minimum	42.950 ms
Maximum	234.476 ms

Table 7.2: Round Trip Time

7.2.2 Steganographic expansion

7.2.3 Goodput

In order to see how fast data can travel through the BitSmuggler pipe, we measure the goodput (application level throughput) of the system and compare it to the performance of an HTTP connection to an nginx server.

The experiments consist of downloads from server to client of files of varying sizes.

7.2.4 Resilience

should i still do this. it is a bit more complicated

Chapter 8

Conclusion

Chapter 9

Annexes

C code for ballparking how much computation can be performed in 1 microsecond. Compiled with g++ -Ofast.

```
#include <stdio.h>
#include <sys/time.h>


long long current_timestamp() {
    struct timeval te;
    gettimeofday(&te, NULL); // get current time
    return te.tv_usec;
}

int compute(int x) {
  return (x + 1) *(x + 1);
}

int main() {
  int s = 0;
  long long start = current_timestamp();
  int n = 10000;
  char a[n];
  for (int i = 0; i < n; i++)
    s += compute(a[i]);
  printf("squaring an array of \%d bytes takes \%llu microseconds", n,
      current_timestamp() - start);
}
```

Bibliography

Appelbaum. Technical analysis of the ultrasurf proxying software. 2012. URL https://media.torproject.org/misc/2012-04-16-ultrasurf-analysis.pdf.

D. J. Bernstein. A state-of-the-art diffie-hellman function, March 2006. URL http://cr.yp.to/ecdh.html.

bittorrent.org. Introduction to bittorrent, 2014. URL http://www.bittorrent.org/introduction.html.

blockedinchina. Test if any website is blocked in china. URL http://www.blockedinchina.net/.

brl github repo owner. obfuscated-openssh, 2009. URL https://github.com/brl/obfuscated-openssh.

Guardian Charles Arthur. China tightens great firewall internet control, 2012. URL http://www.theguardian.com/technology/2012/dec/14/china-tightens-great-firewall-internet-control.

Richard Clayton. Ignoring the great firewall of china. 2006.

torrentfreak ernesto. Thunder blasts utorrents market share away, 2009. URL http://torrentfreak.com/thunder-blasts-utorrents-market-share-away-091204/.

hikinggfw.com. hikinggfw: Faq, 2013. URL http://hikinggfw.org/faq/.

Amir Houmansadr, Chad Brubaker, and Vitaly Shmatikov. The parrot is dead: Observing unobservable network communications. 2013. URL http://dx.doi.org/10.1109/SP.2013.14.

Aaron Johnson. Users get routed: Traffic correlation on tor by realistic adversaries. 2013.

Benjamin Eldman Jonathan Zittrain. Empirical analysis of internet filtering in china. 2003. URL http://cyber.law.harvard.edu/filtering/china/appendix-tech.html.

Oleg Kyseliov. Eliminating array bound checking through non-dependent types. 2004. URL http://okmij.org/ftp/Haskell/types.html#branding.

G Lowe. The great dns wall of china. 2007.

Hooman Mohajeri Moghaddam, Baiyu Li, Mohammad Derakhshani, and Ian Goldberg. Skypemorph: Protocol obfuscation for tor bridges. 2012. URL http://doi.acm.org/10.1145/2382196.2382210.

Roberto G. Cascella Mohamad Jaber and Chadi Barakat. Can we trust the inter-packet time for traffic classification? 2011. URL http://www-sop.inria.fr/members/Chadi.Barakat/ICC2011.pdf.

JEFF LARSON NICOLE PERLROTH and SCOTT SHANE. N.s.a. able to foil basic safeguards of privacy on web, 2013. URL N.S.A.AbletoFoilBasicSafeguardsofPrivacyonWeb.

OpenNet. Opennet: Research, 2013. URL https://opennet.net/research.

Stefan Lindskog Philipp Winter. Spoiled onions: Exposing malicious tor exit relays. 2013. URL http://dx.doi.org/10.1109/SP.2013.14.

phobos. Iran partially blocks encrypted network traffic. URL https://blog.torproject.org/blog/iran-partially-blocks-encrypted-network-traffic.

Tor Project. Tor: Bridges, 2011a. URL https://www.torproject.org/docs/bridges.html.en.

Tor Project. Tor: Overview, 2011b. URL https://www.torproject.org/about/overview.html.en.

Tor Project. Tor pluggable transports wiki page, 2011c. URL https://trac.torproject.org/projects/tor/wiki/doc/PluggableTransports.

psiphon.ca. Psiphon website, 2014. URL http://psiphon.ca/index.php.

Sandvine. Global internet phenomena report 1h 2013. URL https://www.sandvine.com/downloads/general/global-internet-phenomena/2013/sandvine-global-internet-phenomena-report-1h-2013.pdf.

Michael Snoyberg. Preliminary warp cross-language benchmarks, March 2011. URL http://www.yesodweb.com/blog/2011/03/preliminary-warp-cross-language-benchmarks.

Sun Tzu. The art of war.

ultrareach.us. Ultrareach website, 2014. URL https://ultrasurf.us/.

Z. Weinberg. Stegotorus: A camouflage proxy for the tor anonymity system. 2012.

Tim Wilde. Great firewall tor probing, 2011. URL https://gist.github.com/twilde/da3c7a9af01d74cd7de7.

Brandon Wiley. Dust, 2013. URL https://github.com/blanu/Dust.

Philip Winter. How the great firewall of china is blocking tor year. 2013a.

Philipp Winter. Scramblesuit: A polymorphic network protocol to circumvent censorship. 2013b. URL http://www.cs.kau.se/philwint/scramblesuit/.

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